

System 35 Desktop Computer

Assembly Development ROM



HEWLETT-PACKARD

HEWLETT  PACKARD

Warranty Statement

Hewlett-Packard products are warranted against defects in materials and workmanship. For Hewlett-Packard Desktop Computer Division products sold in the U.S.A. and Canada, this warranty applies for ninety (90) days from date of delivery.* Hewlett-Packard will, at its option, repair or replace equipment which proves to be defective during the warranty period. This warranty includes labor, parts, and surface travel costs, if any. Equipment returned to Hewlett-Packard for repair must be shipped freight prepaid. Repairs necessitated by misuse of the equipment, or by hardware, software, or interfacing not provided by Hewlett-Packard are not covered by this warranty.

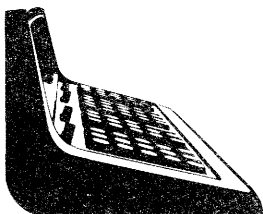
NO OTHER WARRANTY IS EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR CONSEQUENTIAL DAMAGES.

*For other countries, contact your local Sales and Service Office to determine warranty terms.

Assembly Development ROM



HP 9835A Desktop Computer



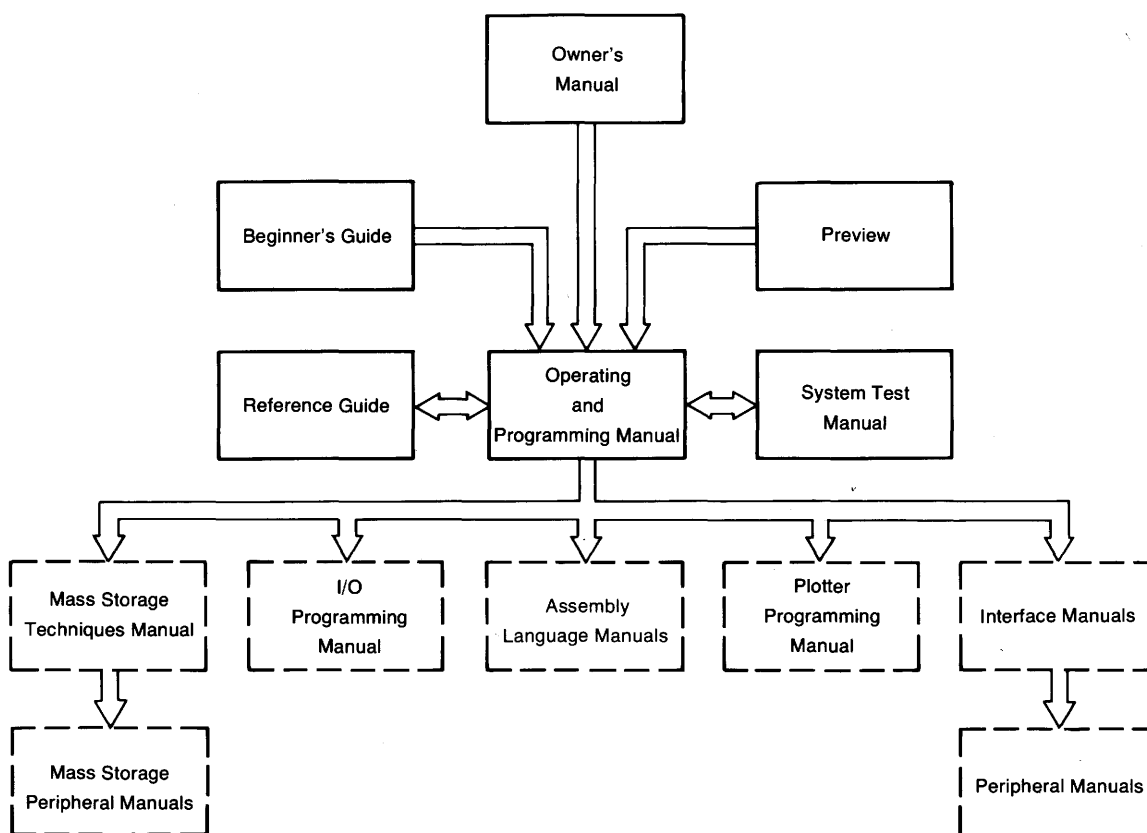
Hewlett-Packard Desktop Computer Division
3404 East Harmony Road, Fort Collins, Colorado 80525
(For World-wide Sales and Service Offices see back of manual.)
Copyright by Hewlett-Packard Company 1979

Customer Questionnaire

To help us in preparing new manuals, there is a questionnaire in the back of this manual. Your answers to the questions can assist in producing better, more useful manuals. Your feedback is our only way of knowing the validity of our manuals. Please complete the questionnaire and mail it — postage is already paid in the United States. Thank you.

System 35 Manual Reference

The following block diagram shows manuals that are included in the System 35 Documentation scheme and suggested progression. Dotted-line borders indicate those manuals are available with specific options; solid borders indicate those manuals that are shipped with every System 35.



Chapter Summaries

Chapter 1. General Information. An introduction to the product and the manual. The purpose and differences of the two Assembly Language ROMs are explained. ROM installation procedures are given. A glossary is provided, along with a discussion of the syntactical forms used in the manual.

Chapter 2. Getting Started. A general discussion of the assembly language system. A format for the creation of an assembly language program is presented. Topics such as modules, routines, and memory allocation are discussed, along with methods of using them effectively. Also discussed is the storage and retrieval of modules on mass storage.

Chapter 3. The Processor and the Operating System. Necessary information on the structure of the processor and the operating system is presented. Topics covered are: machine architecture, memory organization, data structures and arithmetic, and the machine instructions.

Chapter 4. Assembly Language Fundamentals. The basic statements and syntaxes used throughout the assembly language are discussed. Program entry, assembling, symbolic operations, module creation, program and variable storage, and utilities are the topics covered.

Chapter 5. Arithmetic. Arithmetic operations are reviewed and the arithmetic utilities are discussed. Floating point and BCD arithmetic are explained.

Chapter 6. Communicating between Basic and Assembly Language. The techniques used to pass information to and from the assembly language programs are discussed. Calling assembly routines and passing parameters are presented, along with issues involved with using common. Applicable utilities are also discussed.

Chapter 7. I/O Handling. The various techniques of handling the receiving and sending of information to peripheral devices is presented. Topics are: a review of I/O-type machine instructions, registers, applicable utilities, interrupts and interrupt service routines, handshake-type of I/O, direct memory access, and mass storage devices.

Chapter 8. Debugging. Techniques for isolating and correcting logic problems in assembly programs are discussed. Included in the discussion are techniques for stepping through programs, getting dumps, patching, and using the keyboard.

Chapter 9. Errors and Error Processing. A discussion of Assembly Language ROM and other related errors, and what causes them. Included are methods for trapping errors and possible methods for correcting them.

Table of Contents

Chapter 1: General Information

Structure of the Manual	2
Purpose of the ROMs	2
ROM Installation	3
Buzzwords	4
Fundamental Syntax	6

Chapter 2: Getting Started

Developing Routines for Later Use	7
Overview	9
Program Creation	9
Program Entry	14
Other Extensions	16
Modules, Routines, and Such	17
Names	17
Survey of Modules and Routines	18
Setting Aside Memory	19
Retrieving and Storing Modules	22

Chapter 3: The Processor and the Operating System

Machine Architecture	25
Registers	26
General Memory Organization	28
Protected Memory	28
Base and Current Page	29
Data Structures	30
Integers	30
Strings	30
Full-Precision Numbers	31
Short-Precision Numbers	31
Machine Instructions	32
Operands	32
Indirect Addressing	34
Load/Store Group	34
Integer Math Group	35
Branch Group	36
Test/Branch Group	37

Test/Alter/Branch Group	38
Shift/Rotate Group	40
Logical Group	41
Stack Group	42
BCD Math Group	44
I/O Group	47
Miscellaneous	48

Chapter 4: Assembly Language Fundamentals

Program Entry	49
Assembly Language Source	51
Actions	51
Labels	51
Comments	53
Syntaxing the Source	53
Creating Modules	55
Storage	56
Modules	56
Variables	56
Data Generators	57
Repeating Instructions	59
Assembling	60
Effect of BASIC Environments	60
Source Listing Control	61
Page Format	62
Page Length	63
End-of-Page Control	63
Page Headings	64
Blank Line Generation	65
Non-Listable Pseudo-Instructions	65
Conditional Assembly	65
Relocation	68
Symbolic Operations	69
Pre-Defined Symbols	69
Defining Your Own	71
Literals	72
Evaluation of Literals	72
Nesting Literals	73
Nonsensical Uses of Literals	74
Literal Pools	74

Expressions	75
External Symbols and Elements	77
Other Absolute Elements	78
Utilities	79

Chapter 5: Arithmetic

Binary Coded Decimal	83
Arithmetic Machine Instructions	84
BCD Registers	84
BCD Arithmetic	84
Addition	85
Ten's Complement for BCD	86
Floating Point Summations	88
Normalization	89
Rounding	89
Floating Point Multiplication	90
Floating Point Division	92
The FDV Instruction	94
Thirteen-Digit Dividends	95
Floating-Point Division Example	96
Arithmetic Utilities	99
Utility: Rel—math	99
Utility: Rel—to—int	102
Utility: Rel—to—sho	103
Utility: Int—to—rel	104
Utility: Sho—to—rel	105

Chapter 6: Communication Between BASIC and Assembly Language

The ICALL Statement	107
Corresponding Assembly Language Statements	108
Arguments	109
“Blind” Parameters	112
Getting Information on Arguments	113
Utility: Get—info	114
Retrieving the Value of an Argument	116
Utility: Get—value	117
Utility: Get—element	118
Utility: Get—bytes	119
Utility: Get—elem—bytes	120

Changing the Value of an Argument	122
Utility: Put—value	122
Utility: Put—element	123
Utility: Put—bytes	124
Utility: Put—elem—bytes	125
Using Common	127
Busy bits	130
Utility: Busy	131

Chapter 7: I/O Handling

Peripheral-Processor Communication	133
Interfaces	134
Registers	134
Select Codes	134
Status and Control Registers	136
Status and Flag Lines	137
Programmed I/O	138
Interrupt I/O	138
Priorities	140
Interrupt Service Routines and Linkage	140
Access	141
Utility: Isr—access	143
State Preservation and Restoration	145
Indirect Addressing in ISRs	146
Direct Memory Access (DMA)	147
DMA Registers	148
DMA Transfers	149
BASIC Branching on Interrupts	150
ON INT Statement	150
Signalling	151
Additional Pre-Defined Symbols	153
Prioritizing ON INT Branches	153
Environmental Considerations	155
Disabling ON INT Branching	156
Mass Storage Activities	156
Reading from Mass Storage	157
Utility: Mm—read—start	158
Utility: Mm—read—xfer	159

Writing to Mass Storage	160
Utility: Mm—write—start	161
Utility: Mm—write—test	161
System File Information	163
Utility: Get—file—info	164
Utility: Put—file—info	165
Printing	166
Utility: Printer_select	166
Utility: Print_string	167

Chapter 8: Debugging

Stepping Through Programs	170
Individual Instruction Execution	170
Setting Break Points	174
Simple Pausing	174
Transfers	175
Environments	176
Data Locations	177
IBREAK Everywhere	178
Number of Break Points	179
Clearing Break Points	179
Interrogating Processor Bits	180
Protected Memory	180
Dumps	181
Value Checking	183
Functions	184
DECIMAL	184
OCTAL	184
IADR	185
IMEM	186
Patching	187

Chapter 9: Errors and Error Processing

Types of Errors	189
Syntax-Time and Assembly-Time Errors	189
Run-Time Errors	190
Utility: Error_exit	191
Run-Time Messages	193
Assembly-Time Messages	195

Appendix A: ASCII Character Set	
ASCII Character Codes	204
Appendix B: Machine Instructions	
Detailed List	207
Bit Patterns and Timings	221
Alphabetic List	221
Approximate Numerical List	221
Appendix C: Pseudo-Instructions	223
Appendix D: Assembly Language BASIC Language Extensions Formal Syntax	225
Appendix E: Pre-Defined Assembler Symbols	231
Appendix F: Utilities	233
Appendix G: Writing Utilities	235
Appendix H: I/O Sample Programs	
Handshake String Output	237
Handshake String Input	239
Interrupt String Output	241
Interrupt String Input	244
DMA String Output	247
DMA String Input	250
HP-IB Output/ Input Drivers	253
Real-Time-Clock Example	257
Appendix I: Demonstration Cartridge	
Using the tape	261
Typing Aids	261
Appendix J: Error Messages	265
Mass Storage ROM Errors	269
Plotter ROM Errors	269
Assembly Language ROM Errors	270
Assembly Time Errors	271
Appendix K: Maintenance	
Maintenance Agreements	273
Sales & Service Offices	274
Subject Index	277

Chapter 1

Table of Contents

General Information

Structure of the Manual	2
Purpose of the ROMs	2
ROM Installation	3
Buzzwords	4
Fundamental Syntax	6

Chapter 1

General Information

Welcome to the world of assembly language programming on the 9835A/B.

It is the design of the Assembly Development Read Only Memory (ROM) to help extend the capabilities of your 9835A/B by giving you greater control and speed through the use of machine instructions, pseudo-instructions, and extensions to the BASIC language.

The assembly language system is provided to you as ROMs which plug into the drawers provided for that purpose in the 9835A/B. There are three physical ROMs, comprising two "logical" ROMs —

- The Assembly Development ROM. Two physical ROMs. This ROM is always provided with an Execution ROM (together comprising HP product number 98339A), and the three ROMs as a unit constitute the assembly language system of the 9835A/B.
- The Assembly Execution ROM, HP product number 98338A. One physical ROM. Since this ROM is an integral part of the assembly language system, the use of the capabilities in this ROM is incorporated into the discussions in this manual. Information on this ROM can be found separately in the Assembly Execution ROM manual (HP part number 09835-90082).

It is assumed throughout this manual that you are familiar with the basic operation and language of the 9835A/B. It is also assumed that you are reasonably well-acquainted with at least one other assembly language.

Structure of the Manual

It is the intent of this manual that you should be able to find between its covers everything you need to know to use the assembly language effectively. However, since assembly language programming is a complex topic, the manual relies a great deal on your past experience. Most of the information is in succinct presentations of a particular topic; it is not the intent to “teach” assembly language programming to someone not familiar with the topic.

The major topics covered are: assembly language program creation (Chapter 2), the processor and relevant operating system constructs (Chapter 3), assembly language fundamentals (Chapter 4), arithmetic (Chapter 5), communications with BASIC (Chapter 6), I/O handling (Chapter 7), debugging tools (Chapter 8), errors and error processing (Chapter 9). Each topic, or chapter, has a summary at the beginning detailing the information to be presented therein. A compilation of these summaries can be found immediately preceding the Table of Contents.

The manual is organized so that each topic can be covered completely within a given chapter. This approach was chosen over the strict syntactical or semantical treatment of the individual statements and instructions. As a consequence, you may find this difficult to use as a “quick reference” for syntax and meaning of the individual commands.

To meet your needs for “quick reference” material, an Assembly Language System Quick Reference Manual (HP part number 09835-90081) is provided. In addition, you will find much of the information in this manual condensed and tabulated in the various appendices of this manual.

A recommended method for using the manuals is to start with this one as your basic learning tool. Then you should be able to use the Quick Reference Manual effectively for all future reference.

Purpose of the ROMs

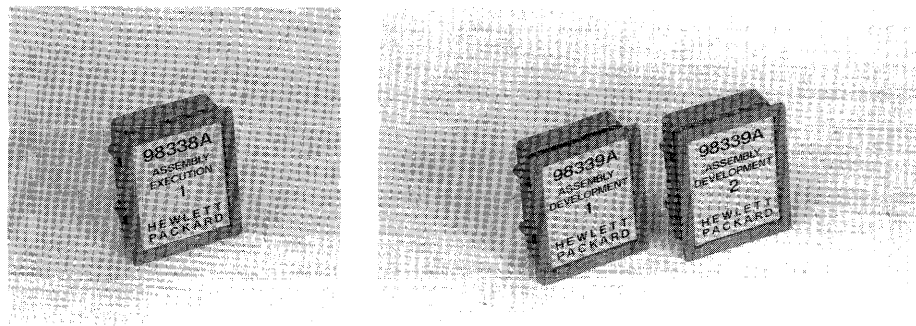
The Development ROM is used to write and debug assembly language programs on the 9835A/B. The Execution ROM, provides the capability to load, run, and store assembled routines and modules.

The Execution ROM can be used independently of the Development ROM. However, the Development ROM cannot be used without the Execution ROM. The latter’s capabilities, therefore, are considered in this manual as an inherent part of the Development ROM. Because of the overhead required by the debugging features provided by the Development ROM, programs run more rapidly if the Execution ROM is used without the Development ROM.

ROM Installation

Before assembly language programming can proceed, the ROMs must be in place. The installation is a simple process.

There are several ROM drawers for the computer: one on the right side of the machine and four in front. Each front drawer holds up to four ROMs; the side drawer holds up to fourteen. ROMs may be placed in any ROM slot in any drawer.



Assembly Language System ROMs

To add the ROMs, turn off the computer and remove a ROM drawer (by pulling outwards on it until it is completely separated from the computer). Insert the ROMs, one at a time, following this procedure: you should orient the ROM so that its label reads the same way as the others in the drawer (with the bottom of the lettering toward the “front” of the drawer). Then insert it vertically in one of the unused slots. Make sure that it slides in all the way to the bottom of the connector. There are small raised ribs on both sides of each ROM which will fit into recesses in the slot; if the ribs don’t fit, you have not oriented the ROM correctly.

After inserting both ROMs, re-insert the drawer in the machine (by pushing on it until it is flush with the outside cover of the machine). With this done, you are now ready to begin writing assembly language programs.

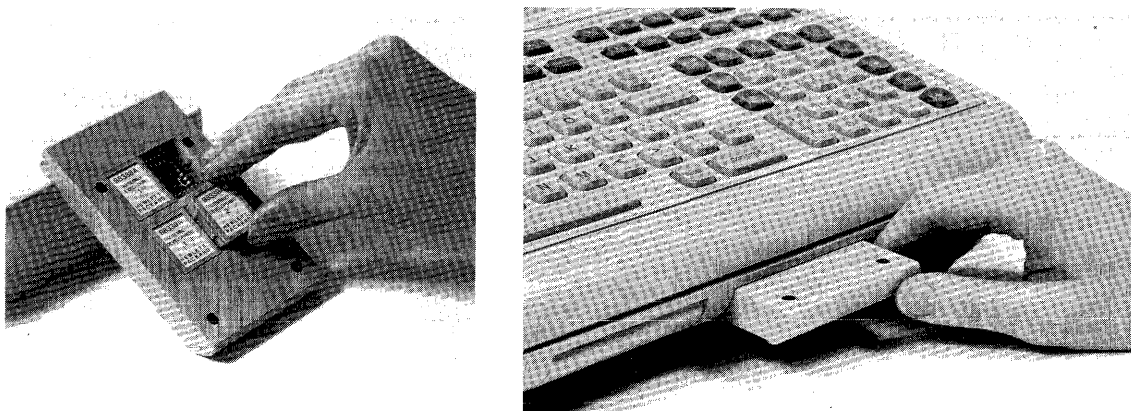


Figure 1. Installing the Development ROM

Buzzwords

During the course of the discussions in this manual, words and phrases are used which are in common circulation among those who are familiar with assembly languages. While the meaning of most are either well-known, or are deducible from the context, there are a few which may be unfamiliar, or unique to the 9835A/B assembly language, or are variable from one assembly language to the next and thus need to be defined for this one. They are —

assembled location — a reference to a location in memory which may be specified in one of the following forms —

```
{symbol} [ , {numeric expression} ]
{expression} [ , {numeric expression} ]
```

where:

{symbol} is an assembly location. It may be either a label for a particular machine instruction (in which case the address of the associated instruction is used), or an assembler-defined symbol (in which case the associated absolute address is used), or a symbol defined by an EQU instruction (described in the “Symbolic Operations” of Chapter 4).

{expression} may be a numeric expression or a string expression. If numeric, a decimal calculation is performed and the result is interpreted as an octal value; if the result is not an octal representation or an integer, an error results. If a string expression is used, the string must be interpretable as either an octal integer constant or a known assembly symbol (see {symbol} above).

{numeric expression} serves as a decimal offset from the given label or constant.

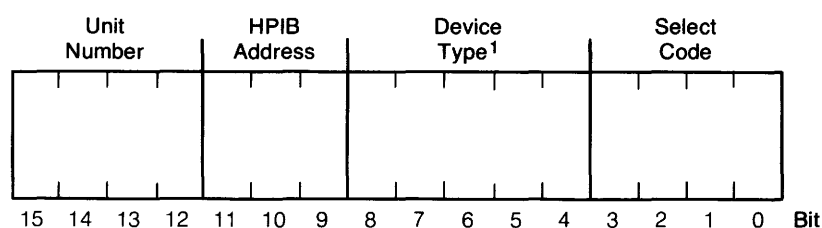
byte — a group of 8 binary digits (bits).

busy bits — each variable located in the BASIC value or common areas has associated with it two bits: a “read” busy bit, and a “write” busy bit. When a busy bit is set, all attempts to perform the associated function on that variable are locked out. When a busy bit is cleared, the function may be performed on the variable.

conditional assembly — an assignation that certain portions of a module are not to be assembled unless a condition has been set. The portions begin with any of the IFA through IFH, and IFP, pseudo-instructions, and end with the next XIF pseudo-instruction. IFA uses the A-condition as a test, and so on. The conditions are set by the statement assembling the module (IASSEMBLE).

interrupt service routine (ISR) — an assembly language routine intended to perform a certain action, or set of actions, when the computer receives a request from an external device. An “active” ISR is one which is currently enabled for a given device.

mass storage unit specifier (msus) — a single word corresponding to the BASIC language mass storage unit specifier as described in either the 9835A/B Operating and Programming Manual — HP part number 09835-90000 — or the Mass Storage Techniques Manual — HP part number 09835-90070. An msus has the following structure —



An msus can designate the current default as its mass storage device (meaning it will use the device indicated by the last MASS STORAGE IS statement executed). This is designated by having the msus be all ones (i.e., equal to -1).

object module — a section of assembled code stored in the particular region of memory set aside for it. Though the source module for the object code may no longer be resident in memory, when created, the module was delimited by certain pseudo-instructions (NAM and END) and is referenced by the name given to it by the NAM pseudo-instruction.

octal expression — a numeric expression which, when displayed or printed, appears as an octal (base-8) number. Within arithmetic operations, it has a decimal value (base-10). Thus, the value 17_8 will appear as 17 (representing the value 15_{10}), but if arithmetic was performed on it, it would act as if it were 17_{10} . All octal expressions are necessarily integers in the range of 0 to 177777_8 .

source module — a section of assembly language source code beginning with a NAM pseudo-instruction and ending with the END pseudo-instruction.

word — two bytes; a group of 16 binary digits (bits).

¹ The device type is the ASCII code for the type minus 1008.

Fundamental Syntax

The syntax conventions used in this manual are those used in the Operating and Programming Manual for the 9835A/B —

`dot matrix` All syntax items displayed in dot matrix form should be programmed as shown.

[] Items contained in brackets are optional items.

... Ellipses mean that the previous item may be repeated indefinitely.

In addition, the following convention is employed throughout the Assembly Language series of manuals —

{ } Items contained in braces are syntax items considered as a unit. The names inside are usually descriptive of the function intended for that item. Whenever an item enclosed in braces appears in the text, the notation refers to the same notation within an earlier syntax.

Chapter 2

Table of Contents

Getting Started

Developing Routines for Later Use	7
Overview	9
Program Creation	9
Program Entry	14
Other Extensions	16
Modules, Routines, and Such	17
Names	17
Survey of Modules and Routines	18
Setting Aside Memory	19
Retrieving and Storing Modules	22

Chapter 2

Getting Started

Summary; This chapter contains a general discussion of the assembly language system. A format for the creation of an assembly language program is presented. Topics such as modules, routines, and memory allocation are discussed, along with methods of using them effectively. Also discussed is the storage and retrieval of modules on mass storage.

The thing to remember about the assembly language system is that it has been thoroughly integrated into the operating system of the 9835A/B. Once the ROMs have been installed, you are able immediately to begin programming in assembly language. In addition, you have the capability to load and store your programs on mass storage, to assemble them separately or leave them in source form, to execute them from BASIC and pass BASIC variables to them, and to debug them, including a full pausing and stepping capability.

Developing Routines for Later Use

Most assembly language programs are written with the intent that they will be used many times, not just at the time they are written. It is for just such program development that the full capabilities of the assembly language system come into play. The development comes in several stages. Each stage has its unique requirements and the tools to meet those requirements.

The first stage is creation of the source program. This is achieved by the use of the editing capabilities of the 9835A/B. Additionally, the basic mass storage capabilities of the computer can be used.

The second stage is the creation of the object (or machine) code. This requires not only an assembly of the source, but the ability to allocate special locations in memory to hold the newly created object code.

The third stage is the validation of the routines as written, commonly known as "debugging". This is enabled by calls from a BASIC driver, followed by application of various debugging tools provided by the assembly system. The capabilities to pause and step a program have been extended to assembly language instructions to assist this process.

The fourth stage is to store away the debugged object code so that it may be used at a later time. A special mass storage statement is provided by the assembly language system. This statement stores object code into a special assembly file.

Finally, the end-user of the routines must be able to retrieve the object code from mass storage as it is needed. He also must be able to access the routines from BASIC programs. Both these needs are met with the Execution ROM, so the capabilities are not only provided, but they are provided independent of the program development capabilities located in the Development ROM.

Each of the topics involved in these stages is discussed at length in this manual.

Figure 2 presents a graphical presentation of this overview.

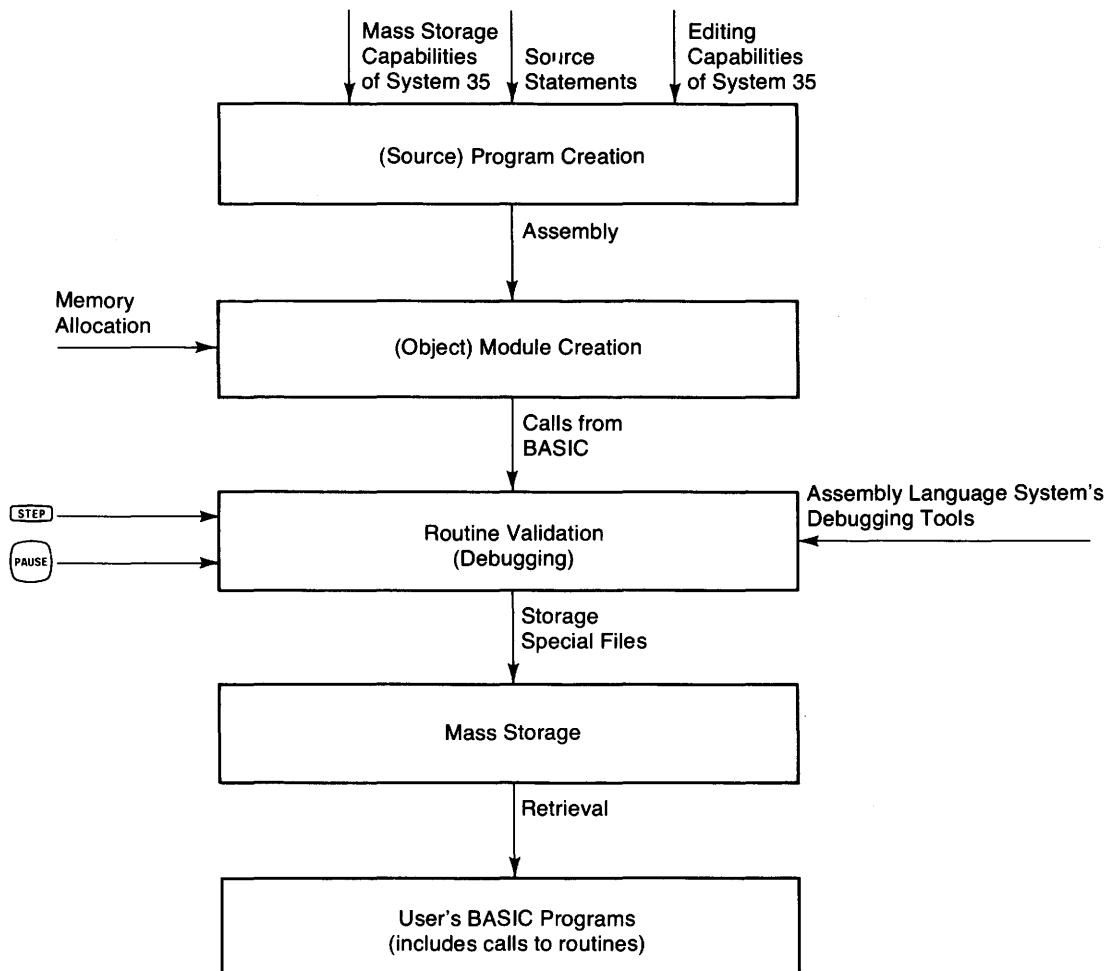


Figure 2. Overview of Assembly Language Routine Development Process

Overview

At this point, there are three fundamental structures to be explained: programs, modules, and routines.

A **program** is the set of source statements from which the object (or machine) code is generated. The assembly source statements are extensions to the BASIC language which is used in the 9835A/B. The statements themselves are stored in the machine as part of the BASIC program in which they reside. At some point, you must take the assembly source statements and assemble them into object code, in order that they can be run. The object code is stored in a specified location in the machine.

A **module** is a subset of the object code. It is a means of separating and identifying parts of the code so that those parts may be used individually (as in mass storage operations). There may be any number of modules present at any one time, limited only by the amount of memory allocated for object code.

A **routine** is a “callable” section of a module. It is analogous to the subprogram in BASIC. It has a named entry point, possibly a parameter list, and (if programmed correctly) a return. A module may contain any number of routines, again limited only by the amount of memory allocated to hold the object code.

In short, the usefulness of each structure is as follows —

- **Programs** contain assembly language source code.
- **Modules** contain object code to be loaded from or stored on mass storage.
- **Routines** are executable sections of object code.

Program Creation

The first matter which is likely to concern you about the assembly language system is how to create an assembly language program.

In general, the process of creating an assembly language subprogram consists of the following steps —

1. Enter and store the source code (program).
2. Create an area in memory which will ultimately contain the object code.

3. Assemble the source code into object code, storing the latter into the area of memory set aside for it.
4. Execute the object code (routines) from BASIC “drivers”.

Each of these steps will be discussed at length in the pages of this manual, along with a number of not-so-incidental side-topics (such as “debugging” techniques). The purpose of **this** short section is to give you an impression of the general procedure through which an assembly language subprogram is created.

As an example to use to demonstrate the process, suppose the following task has been assigned to you —

Requirement: Write an assembly language subprogram which takes two integer values and multiplies them together as integers. If the result overflows the range of an integer (– 32 768 to + 32 767), then the subprogram should return the same error as the system would (i.e., error number 20).


With this task in hand, suppose that you have completed a programming analysis that suggests that the following assembly language source code would fulfill the subprogram’s functions —¹

```

NAM Multiplication ! Beginning of module
EXT Error_exit,Get_value,Put_value ! Utilities
Integers: BSS 2      ! Storage area for integers created
SUB       ! Indicates entry point follows
Input1:   INT       ! Indicates "integer parameters are
Input2:   INT       ! passed in the order given by these
Output:   INT       ! statements and are given names
Multiply: LDA =Integers ! Actual entry point (name: Multiply);
          LDB =Input1   ! routine begins by fetching actual
          JSM Get_value ! value of the input parameters
          LDA =Integers+1 ! from BASIC and storing them where
          LDB =Input2   ! the routine can use them
          JSM Get_value
          LDA Integers  ! Then it loads the values into the
          LDB Integers+1 ! arithmetic accumulator and
          MPY          ! finally multiplies them
          SBP ++2      ! A check for overflow is performed
          CME         ! by checking the result for anything
          SZB ++3      ! in the B register when it should be 0
          LDA =20      ! and if it isn't, Error 20 is selected
          JSM Error_exit ! and the routine is aborted
          STA Integers  ! If everything is OK, then result stored
          LDA =Integers ! The product is then returned to the
          LDB =Output   ! output variable in BASIC listed
          JSM Put_value ! among the arguments
          RET 1         ! We're finished, so return to BASIC
END Multiplication ! End of module

```

¹ The fact that it is rarely possible to create a running program at this stage should not get in the way of accepting the example. Usually there is debugging involved in later stages.

Now that the routine has been developed, it is necessary to get it into the memory of the machine as a program. This is done by preceding each and every assembly language statement with the keyword `ISOURCE` and entering it as a program line. The process of entering (with the keyword included) is the same as with any other BASIC statement — so you can use `EDIT` or `AUTO` and the  key in the same way you normally enter any BASIC statement. (This process is fully described in the “Program Entry” section of this chapter.)

The final result of entering the routine would look something like —

```

10 ISOURCE      NAM Multiplication ! Beginning of module
20 ISOURCE      EXT Error_exit,Get_value,Put_value ! Utilities
30 ISOURCE      Integers: BSS 2      ! Storage area for integers created
40 ISOURCE      SUB              ! Indicates entry point follows
50 ISOURCE      Input1:  INT        ! Indicates "integer parameters are
60 ISOURCE      Input2:  INT        !   passed in the order given by these
70 ISOURCE      Output:  INT        !   statements and are given names
80 ISOURCE      Multiply: LDA =Integers ! Actual entry point (name: Multiply);
90 ISOURCE      LDB =Input1         !   routine begins by fetching actual
100 ISOURCE      JSM Get_value      !   value of the input parameters
110 ISOURCE      LDA =Integers+1    !   from BASIC and storing them where
120 ISOURCE      LDB =Input2        !   the routine can use them
130 ISOURCE      JSM Get_value
140 ISOURCE      LDA Integers       ! Then it loads the values into the
150 ISOURCE      LDB Integers+1     !   arithmetic accumulator and
160 ISOURCE      MPY                !   finally multiplies them
170 ISOURCE      SBP **2            ! A check for overflow is performed
180 ISOURCE      CMB                !   by checking the result for anything
190 ISOURCE      SZB **3            !   in the B register when it should be 0
200 ISOURCE      LDA =20            !   and if it isn't, Error 20 is selected
210 ISOURCE      JSM Error_exit     !   and the routine is aborted
220 ISOURCE      STA Integers       ! If everything is OK, then result stored
230 ISOURCE      LDA =Integers      ! The product is then returned to the
240 ISOURCE      LDB =Output        !   output variable in BASIC listed
250 ISOURCE      JSM Put_value      !   among the arguments
260 ISOURCE      RET 1              ! We're finished, so return to BASIC
270 ISOURCE      END Multiplication ! End of module

```

This source code demonstrates the three critical items in assembly subprograms. First, a routine has to be part of a module; modules are delimited with the `NAM` and `END` pseudo-instructions (see lines 10 and 270 in the source). Second, a routine has to have an entry point; this consists of a `SUB` pseudo-instruction (see line 40), any parameters (see lines 50 through 70), and a name (the label used on the first machine instruction following the `SUB`, see line 80). Finally, a routine must be able to return to the BASIC program which called it; this is accomplished with the `RET 1` instruction (see line 260).

The `NAM`, `END`, and `SUB` pseudo-instructions are discussed in Chapter 4. The `RET 1` instruction is discussed in Chapter 3.

12 Getting Started

The next three steps in program creation are each satisfied with BASIC-executable statements. Creation of a storage area for the object code for the program (which can be estimated at less than 40 words; there is essentially one word of object code per line of source) is accomplished by programming the statement —

```
280 ICOM 40
```

(The ICOM statement is fully discussed in the “Setting Aside Memory” section of this chapter.)

This can be followed in the same program by an instruction to assemble the source code into object code —

```
290 IASSEMBLE Multiplication
```

(The IASSEMBLE statement is fully discussed in Chapter 4.)

If the assembly is successful (and it will be in this example), then the routine can be called and used as desired. A typical call looks like —

```
600 ICALL Multiply(Index,Dimension,Subscript)
610 Array(Subscript)=Value
```

(The ICALL statement is fully discussed in Chapter 6.)

Thus, the final result could easily be —

```
10  ISOURCE      NAM Multiplication ! Beginning of module
20  ISOURCE      EXT Error_exit,Get_value,Put_value ! Utilities
30  ISOURCE      Integers: BSS 2      ! Storage area for integers created
40  ISOURCE      SUB                      ! Indicates entry point follows
50  ISOURCE      Input1:  INT          ! Indicates "integer parameters are
60  ISOURCE      Input2:  INT          !   passed in the order given by these
70  ISOURCE      Output:  INT          !   statements and are given names
80  ISOURCE      Multiply: LDA =Integers ! Actual entry point (name: Multiply);
90  ISOURCE      LDB =Input1           !   routine begins by fetching actual
100 ISOURCE      JSM Get_value         !   value of the input parameters
110 ISOURCE      LDA =Integers+1       !   from BASIC and storing them where
120 ISOURCE      LDB =Input2           !   the routine can use them
130 ISOURCE      JSM Get_value
140 ISOURCE      LDA Integers          ! Then it loads the values into the
150 ISOURCE      MPY Integers+1        !   arithmetic accumulator and
160 ISOURCE      SBP **2               !   finally multiplies them
170 ISOURCE      CMB                  ! A check for overflow is performed
180 ISOURCE      SZB **3               !   by checking the result for anything
190 ISOURCE      LDA =20               !   in the B register when it should be 0
200 ISOURCE      JSM Error_exit        !   and if it isn't, Error 20 is selected
210 ISOURCE      STA Integers          !   and the routine is aborted
220 ISOURCE
```

```

230 ISOURCE          LDA =Integers      ! The product is then returned to the
240 ISOURCE          LDB =Output        ! output variable in BASIC listed
250 ISOURCE          JSM Put_value      ! among the arguments
260 ISOURCE          RET 1              ! We're finished, so return to BASIC
270 ISOURCE          END Multiplication ! End of module
280 ICOM 40
290 IASSEMBLE Multiplication
.
.
.
600 ICALL Multiply(Index,Dimension,Subscript)
610 Array(Subscript)=Value
.
.
.

```

It isn't necessary that a program be assembled in every BASIC program which uses it. Object code can be stored on mass storage with a statement like —

```
300 ISTORE Multiplication;"MULT"
```

So if the example were instead made to read —

```

10 ISOURCE          NAM Multiplication ! Beginning of module
20 ISOURCE          EXT Error_exit,Get_value,Put_value ! Utilities
30 ISOURCE          Integers: BSS 2      ! Storage area for integers created
40 ISOURCE          SUB              ! Indicates entry point follows
50 ISOURCE          Input1: INT         ! Indicates "integer parameters are
60 ISOURCE          Input2: INT         ! passed in the order given by these
70 ISOURCE          Output: INT        ! statements and are given names
80 ISOURCE          Multiply: LDA =Integers ! Actual entry point (name: Multiply);
90 ISOURCE          LDB =Input1         ! routine begins by fetching actual
100 ISOURCE          JSM Get_value      ! value of the input parameters
110 ISOURCE          LDA =Integers+1    ! from BASIC and storing them where
120 ISOURCE          LDB =Input2        ! the routine can use them
130 ISOURCE          JSM Get_value
140 ISOURCE          LDA Integers       ! Then it loads the values into the
150 ISOURCE          LDB Integers+1     ! arithmetic accumulator and
160 ISOURCE          MPY                ! finally multiplies them
170 ISOURCE          SBP +=2            ! A check for overflow is performed
180 ISOURCE          CMB                ! by checking the result for anything
190 ISOURCE          SZB +=3            ! in the B register when it should be 0
200 ISOURCE          LDA =20            ! and if it isn't, Error 20 is selected
210 ISOURCE          JSM Error_exit     ! and the routine is aborted
220 ISOURCE          STA Integers       ! If everything is OK, then result stored
230 ISOURCE          LDA =Integers      ! The product is then returned to the
240 ISOURCE          LDB =Output        ! output variable in BASIC listed
250 ISOURCE          JSM Put_value      ! among the arguments
260 ISOURCE          RET 1              ! We're finished, so return to BASIC
270 ISOURCE          END Multiplication ! End of module
280 ICOM 40
290 IASSEMBLE Multiplication
300 ISTORE Multiplication;"MULT"
310 END

```

the object code is consequently stored into the file “MULT”.

Later programs can retrieve the object code for use, such as in the following program —

```
10  INTEGER Dimension, Index, Subscript
20  ICOM 40
30  ILOAD "MULT"
   .
   .
   .
600 ICALL Multiply(Index, Dimension, Subscript)
610 Array(Subscript)=Value
   .
   .
   .
```

(Both ISTORE and ILOAD are discussed in the “Retrieving and Storing Modules” section of this chapter.)


Program Entry

The assembly language source statement is an **extension** to the BASIC language used in the 9835A/B. This means that each assembly language statement is entered using a “keyword” — in this case ISOURCE — as a message to the operating system that the line is an assembly language statement.

By looking at an example, you can see what is meant —

```
10  LET A=10
20  LET B=20
30  PRINT A,B
40  ISOURCE  NAM Example
50  ISOURCE  NOP
60  ISOURCE  END Example
70  END
```

Lines 10, 20, 30, and 70, are all recognizable as BASIC statements. The keywords they use — LET, PRINT, and END — direct that certain actions take place. Lines 40, 50, and 60, are all assembly language statements; this was indicated by the ISOURCE keyword used in these lines.

Entering assembly language statements, by using the ISOURCE keyword, is thereby the same process as entering other types of BASIC statements. You may use all of the system editing features that you are used to using in the creation of BASIC programs — EDIT, AUTO, etc. You store each line with the  key, as you would any other BASIC line.

Also, assembly lines do not have to be in any special place in the BASIC program. The above example could be re-arranged as follows —

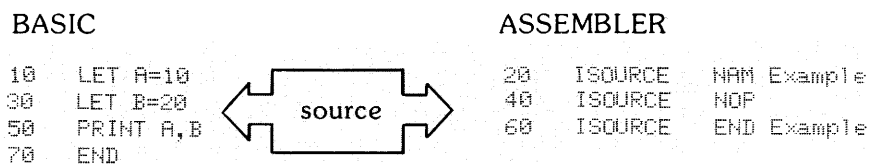
```

10 LET A=10
20 ISOURCE NAM Example
30 LET B=20
40 ISOURCE NOP
50 PRINT A,B
60 ISOURCE END Example
70 END

```

Thus, you are free to enter your assembly statements anywhere in your BASIC program. But, you may ask, what is the effect of spreading them out like this? The answer is, simply, none. When the time comes to use them, assembly statements and BASIC statements are separated by the operating system and treated differently.

When the BASIC program is run, **ONLY** the BASIC statements are executed. The ISOURCE statements are **ignored**, and, as you will be shown in Chapter 4, when the assembly language lines are assembled, the BASIC statements are ignored. A way to consider it is that there are two programs in one — BASIC's and the assembler's. So you can envision the example above as being this way —



You should note, then, that ISOURCE statements are not “executable” in the usual BASIC sense. Their location in the program does not indicate the place where they will be executed. Assembly instructions are not executed until a routine is “called”; this is discussed in detail in Chapter 4.

Now that it has been said that the two types of statements can be thoroughly intermixed, it should also be said that the practice is **not recommended**. As a good programming practice — i.e., for readability and to preserve the self-documenting features of BASIC — it is recommended that assembly statements be collected together and placed in one spot in the program.

The first example is a recommended practice over the second, even though the second is permissible.

Other Extensions

In addition to the ISOURCE statement, there are a number of other BASIC language extensions provided by the assembly language system. Unlike the ISOURCE statement, they are “executable”, and their appearances are part of the BASIC lines (as distinguished from the assembler’s). Where they appear is where the action associated with them is taken. This is identical to the way the other BASIC statements perform. The statements involved are —

- IASSEMBLE
- IBREAK
- ICALL
- ICHANGE
- ICOM
- IDELETE
- IDUMP
- ILOAD
- INORMAL
- IPAUSE OFF
- IPAUSE ON
- ISTORE
- OFF INT
- ON INT

Also provided are four numeric functions —

- DECIMAL
- IADR
- IMEM
- OCTAL

The functions can be used wherever numeric functions in general may be used.

All of these statements (except ICOM and ISOURCE) and the functions are available to you as live keyboard operations as well as programmable statements. A full discussion of each of the statements and functions can be found within this manual.

Modules, Routines, and Such

There are three basic activities associated with using assembled modules and routines. First, there is the need to retrieve them from wherever they may be stored (including providing a place for them to be kept while they are resident in the memory of the machine). Second, there is the actual execution of the routines. And third, there is the occasional requirement to store, or re-store a module on mass storage (including, perhaps, the need to free up the space in memory it previously occupied).

Names

Routines, modules, and files all have names. The names given them may or may not bear some significance to one another; that depends upon you and the way that you name things.

Conventions for the naming of files and methods of general file manipulation can be found in the Operating and Programming Manual and in the Mass Storage Techniques Manual. The conventions are not any different than for files in general.

Names for modules are assigned with the creation of the source. In the assembly language source code, you have a **NAM** pseudo-instruction. This serves two purposes — to designate the beginning of the module and to assign the module a name. All of the assembly source statements which follow the **NAM** are in that module until an **END** pseudo-instruction is encountered. Thus, recalling the previous example —

```
20  ISOURCE  NAM Example
40  ISOURCE  NOP
60  ISOURCE  END Example
```

All of the **ISOURCE** statements between lines 20 and 60 (in this case, just the one) form the module called “Example”. The formal syntaxes of these pseudo-instructions are —

```
NAM {module name}
END {module name}
```

{module name} is a symbol which becomes the name of the module. It follows the same rules as names in BASIC: up to fifteen characters; starts with a capital letter; followed by only non-capital letters, numbers, or the underscore character.

The {module name} in the END statement must correspond to the {module name} of the NAM statement or an assembly error (“EN”) results.

You may have any number of modules in your source code. Each module begins with a NAM and ends with an END pseudo-instruction as above.

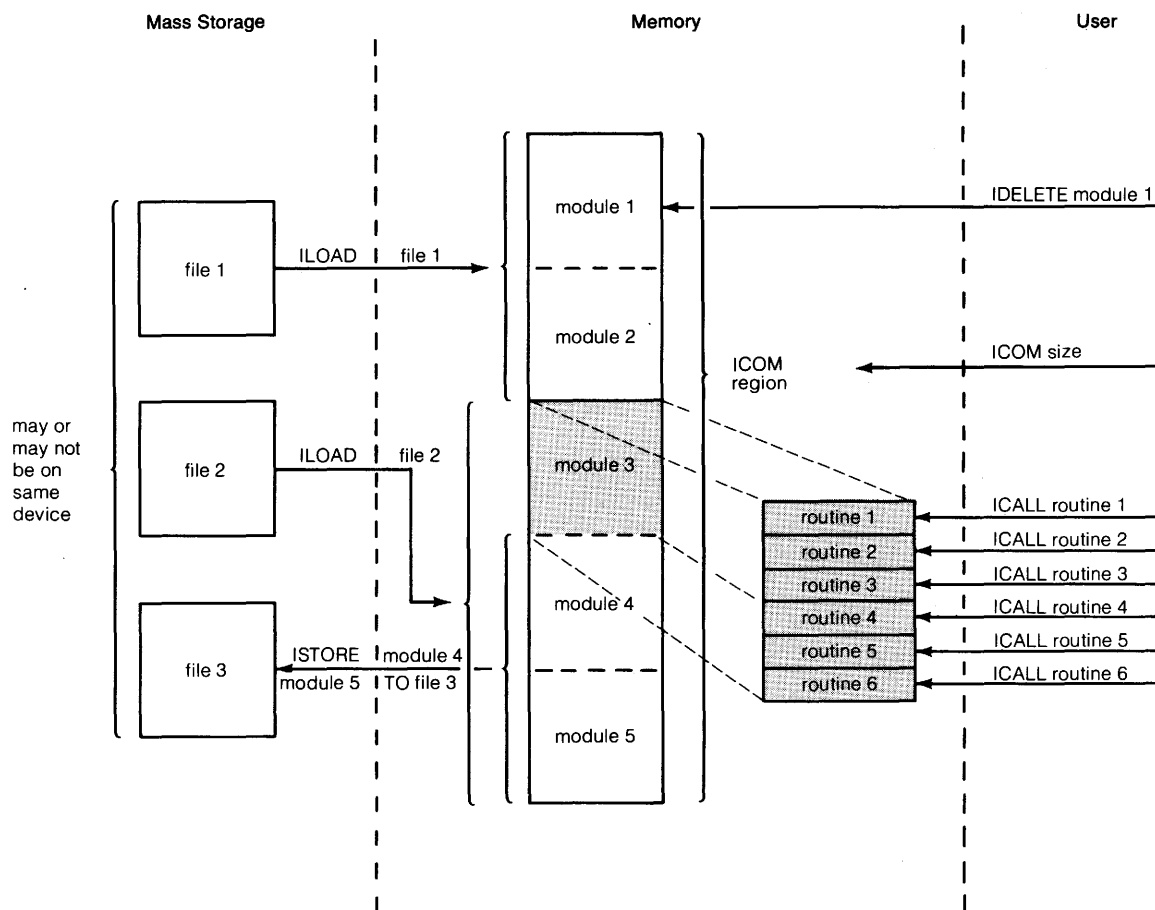


Figure 3. Overview of Routines and Modules.

Survey of Modules and Routines

To sketch the functional relationships of modules and routines, please refer to Figure 3 above.

Modules are stored in files and may be retrieved and placed in memory using the “ILOAD” command. When the ILOAD command is executed, all of the modules in the file are loaded into the memory. Note that many files can be loaded, with many modules each, with all of the modules able to remain resident in the memory.

Alternatively, modules which are already in memory may be stored into a single file using the “ISTORE” command. When the ISTORE command is executed, the designated modules are stored into an “option ROM” (OPRM) type of file (on tape cartridges) or an “Assembly” (ASMB) type of file (on non-tape mass storage media). After storage, the modules are still in memory. They may be removed (i.e., the space they occupy in memory is “freed up”) by using the “IDELETE” command.

The area of memory where the modules are stored is called the “ICOM region”. It is a particular contiguous area which must be large enough to hold all of the object code you wish to have resident in the memory at any one time.

Each module contains one or more routines. Your access to the routines is through the ICALL statement, which is very similar to the CALL statement used for BASIC subprograms. The ICALL statement may have arguments which you need to “pass” (send down) to the routine itself. What these arguments, if any, may be, and what meaning they hold depends upon what you have in mind for that routine. There are corresponding items in the assembly source code; these are discussed in Chapter 6.

Setting Aside Memory

As indicated by Figure 3, you cannot load a module until there is an ICOM region into which to load it. Neither can you assemble your source code into object code unless there is an ICOM region into which the object code can go.

The statement to use to create an ICOM region is —

```
ICOM {size}
```

where {size} is an integer constant indicating the number of words to be used to form the ICOM region. The maximum size is 32 718 words.

The ICOM statement is a “declaration”, that is, it is not executable, but rather is used when assignment of memory takes place just before a program is run. This is similar to a DIM or COM statement. As with a DIM or COM statement, the statement cannot be executed from the keyboard.

Once created, the ICOM region remains in existence until it is explicitly destroyed. But it is possible to change the size by using another ICOM statement.

The order in which modules appear in the ICOM region is determined by the order in which they are loaded using the ILOAD statement discussed in the next section or are created by the IASSEMBLE statement discussed in the next chapter.

In most cases, the space which is freed up by reducing the size of the ICOM region is returned to your available memory space. Sometimes, however, it is not returned, this being caused by the status of the common area allocated in memory, or by other option ROMs. The space is returned whenever —

- There is no common area assigned (with the COM statement); and,
- The requirements of another option ROM do not interfere.

There may be any number of ICOM statements in a program. The current size of the ICOM region is determined by the last one which appears in the program when the RUN key is pressed (or the command RUN is executed).

For example, suppose you have a program with the following statements in it —

```
20  ICOM 984
30  DIM A#[100]
    .
    .
    .
300 ICOM 492
    .
    .
    .
610 ICOM 2000
    .
    .
    .
900 END
```

Upon pressing RUN, the ICOM region would be 2 000 words long. This is because line 610 is the final ICOM appearance.

The region continues to exist even if you load in another program which contains no ICOM statements. All ICOM statements must appear in the **main** program, not in any subprogram.

ICOM statements in a program must appear before any COM statement. This is to insure that the ICOM region will be allocated before the common is allocated.

There are three ways to eliminate the ICOM region —

- Execute SCRATCH A
- Execute ICOM 0 in a program.
- Turn off the machine.

After any of these actions, the region is no longer in existence. If there are any modules in the region, they disappear as well. If any of those modules contain an active interrupt service routine, you get an error (number 193) if you try to eliminate the region using ICOM 0. If any of your routines provided to other users contain active ISRs, your documentation for the routine should warn the users of that fact so they can avoid this error.

The ICOM 0 procedure can be used to assure that all previous modules are deleted. For example, the following sequence —

```
100 ICOM 0
110 ICOM 2000
```

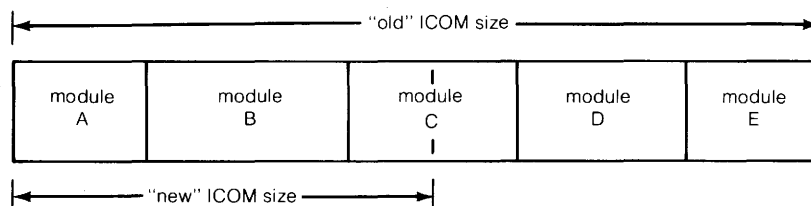
assures that an ICOM region of 2 000 words is in existence at the running of the program, and one completely clear of any previously loaded modules.

When you are altering the size of the ICOM region, the new size specified becomes the size of the region from the moment of running the program. If the size being requested is larger than that which already exists, the additional space needed is requested from the operating system. If the space is available, everything proceeds uneventfully. If the space is not available, an error (number 2) results. To make the space available, one of the following procedures must be followed —

- Execute SCRATCH A.
- Execute SCRATCH C.

Each procedure has its separate effects, and the course selected should be determined by your circumstances at the time. Consult the Operating and Programming Manual for details on the other effects of each of these commands.

If the size being requested is smaller, modules are deleted if they no longer fit into the smaller region. For example, suppose the following situation existed —



Upon compilation of the new ICOM statement, the modules E, D, and C are deleted. None of those modules may contain an active interrupt service routine or an error results (number 193).

Retrieving and Storing Modules

Modules are stored in files on mass storage media as Option ROM (OPRM) or Assembly (ASMB) types of files. On tape media, they are stored in the OPRM type and on non-tape media they are stored in the ASMB type. In this case, the two file types are equivalent.¹

To retrieve a module, or modules, from mass storage, identify the file name of the file containing the module. Combine the name with the mass storage unit specifier² of the device to form a file specifier. Then execute the statement —

```
ILOAD {file specifier}
```

This retrieves ALL the modules in the file and stores them in the ICOM region.

If there are modules already loaded in the ICOM region, these additional modules are added to them, (NOT written over them). If an existing module in the ICOM area has the same name as one of the modules being loaded, the existing module is deleted and the loaded version takes its place.

If you do not want all the modules in a given file, you can purge the unwanted ones from the ICOM region using the IDELETE statement —

```
IDELETE {module name} [{module name} [...]]
```

¹ OPRM-type files may be created by other option ROMs for their particular purposes. In those cases, the contents are entirely different.

² Not to be confused with the single-word msus described in Chapter 1. This form is used by BASIC's Mass Storage statements (see the Operating and Programming Manual or Mass Storage Techniques Manual).

For example, if you had loaded a file which had the routines Larry, Pat, Ed, and Piper, and you want to keep only Larry, then you execute the statements —

```
IDELETE Pat
IDELETE Ed
IDELETE Piper
```

or, more simply —

```
IDELETE Pat, Ed, Piper
```

Deletions do not have to be done immediately after loading. They can be done at any time. After the IDELETE has been executed, the portion of the ICOM region which the module previously occupied is made available for use in loading other modules. The space is NOT returned to the generally available memory; that action is done with an ICOM statement with a smaller size.

Whenever a module is deleted, other modules are moved, as necessary, to take up any slack space in the ICOM region. This is done so that all of the free space in the region is at the end. If a module is being deleted, or being moved as above, and it contains an active interrupt service routine, an error results (number 193).

If you desire at any time to delete all of the modules in your ICOM region, you can do so by executing either of the following statements —

```
IDELETE ALL
IDELETE
```

Sometimes you may desire to move modules in the opposite direction — from memory to mass storage. This is done with the ISTORE statement. The statement has the form —

```
ISTORE {module name} [, {module name} [, ...] ] ; {file specifier}
```

A {module name} must be the name of a module currently stored in the ICOM region. Upon execution of the statement, a file with the name and mass storage unit specifier given in the {file specifier} is created and the modules are stored in the file, in the order listed.

The file created by an ISTORE statement is an OPRM or ASMB type, as appropriate to the medium involved. It can then be used in ILOAD statements at a later time.

In the case that you might want to store all of the routines currently in the ICOM region into a particular file, you can use either of the following statements —

```
ISTORE ALL; {file specifier}  
ISTORE; {file specifier}
```

Chapter 3

Table of Contents

The Processor and the Operating System

Machine Architecture	25
Registers	26
General Memory Organization	28
Protected Memory	28
Base and Current Page	29
Data Structures	30
Integers	30
Strings	30
Full-Precision Numbers	31
Short-Precision Numbers	31
Machine Instructions	32
Operands	32
Indirect Addressing	34
Load/Store Group	34
Integer Math Group	35
Branch Group	36
Test/Branch Group	37
Test/Alter/Branch Group	38
Shift/Rotate Group	40
Logical Group	41
Stack Group	42
BCD Math Group	44
I/O Group	47
Miscellaneous	48

Chapter 3

The Processor and the Operating System

Summary: This chapter contains the necessary information on the structure of the processor and the operating system. Topics covered are: machine architecture, memory organization, data structures, and the machine instructions.

Before proceeding to the actual assembly language, it is useful to discuss the processor and operating system with which you are dealing. This chapter discusses various concepts related to the processor, the machine instruction set, the operating system organization, and data structures.

Machine Architecture

The 9835A/B is developed around a set of processors called a “hybrid”. There are actually three processors — the Binary Processor Chip (BPC), the Input-Output Controller (IOC), and the Extended Math Chip (EMC). Each has its own set of instructions, but all three work in conjunction. It is not necessary in using the assembly system that you know on which chip a particular instruction resides. In the presentation of the instruction set — and for all practical purposes while working with the computer — no distinction need be made between the processors, and the entire instruction set may be considered as being resident on a single processor.

In addition to the processors, the hybrid also contains an I/O bus which is controlled by certain instructions. The I/O bus has an “address” part and a “data” part. Some of the instructions (it is indicated which ones) cause an “input cycle” to occur on the bus, which means that an address is given to the address part of the bus, and the data which appears on the data part is considered to be input. Other instructions cause an “output cycle”, which means that the data is to be output to the given “address”.

Figure 4 is a graphical representation of this architecture.

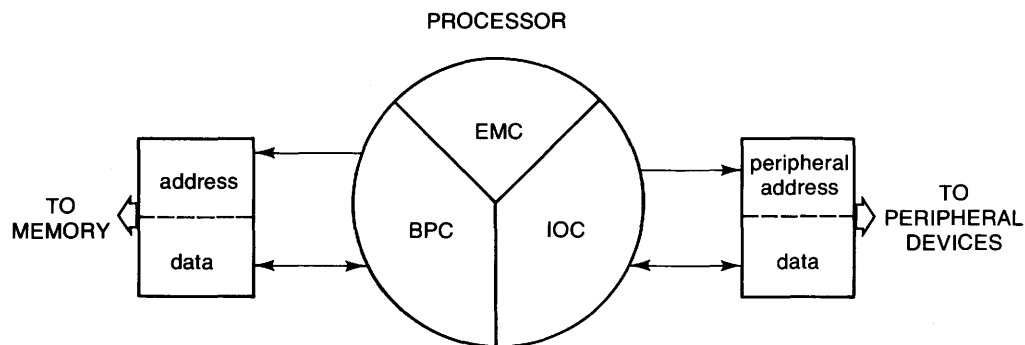


Figure 4. Generalized Machine Architecture

Registers

The memory locations in the machine are addressed from 0 to 177777₈. There are 32 memory locations which are addressed as if they were part of the computer read/write memory, but actually are part of the processor. These locations are called “internal registers”. Each register has a specific location and has been given a name. As you will learn in “Symbolic Operations” (Chapter 4), these names have been reserved and cannot be redefined while using the assembly system.

The internal registers are —

Name	Address (Octal)	Description
A	0	Arithmetic accumulator
Ar2	20-23	BCD arithmetic accumulator
B	1	Arithmetic accumulator
C	16	Stack pointer
Cb	13	Block bit for byte pointer in C (use most significant bit only)
D	17	Stack pointer
Db	13	Block bit for byte pointer in D (use second most significant bit only)
Dmac	15	DMA count register
Dmama	14	DMA memory address register
Dmapa	13	DMA peripheral address register (use lower 4 bits only)
P	2	Program counter
Pa	11	Peripheral address register (use lower 4 bits only)
R	3	Return stack pointer
R4	4	} I/O (Input/Output) registers
R5	5	
R6	6	
R7	7	
Se	24	Shift-extend register

Figure 5 is a map of where these registers lie. In addition to these registers, the addresses 25₈ through 37₈ are also registers, but are not (except for a few isolated cases) used in assembly programming.

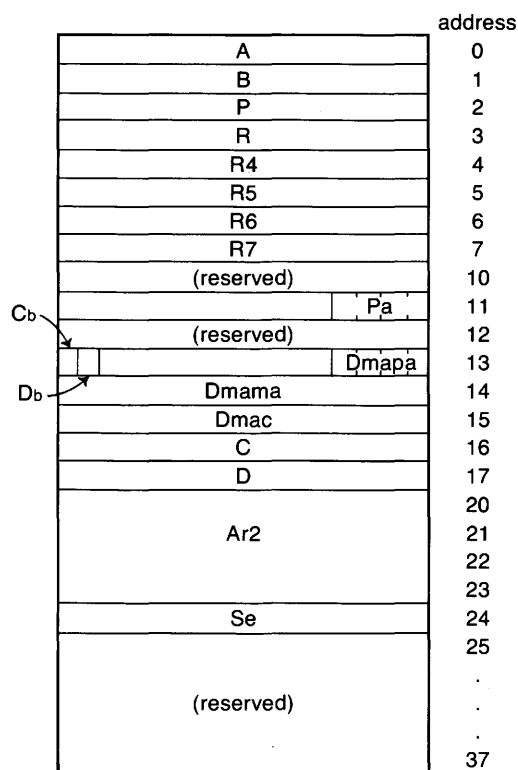


Figure 5. Map of Lowest Memory

All of these registers can be referenced either by their names or by their actual addresses. The two methods are equivalent, though reference by name is recommended as a programming practice.

In addition to the above internal registers, there are some “external” registers which reside in the computer read/write memory. They are —

Name	Address (octal)	Description
Ar1	177770-177773	BCD arithmetic accumulator
Base_page	177620-177701	Base_page temporary area (50 words)
Oper_1	177702	Arithmetic utility operand address registers
Oper_2	177703	
Result	177704	Arithmetic utility result address register

General Memory Organization

In order to find your way around the machine effectively, you should be aware of where things are stored in memory. Occasionally these areas can become considerations in your programming.

First in the memory come the internal registers. They were discussed above.

Next in the memory comes the ICOM area. The starting location is dependent upon system needs, but is always at least 41₈. The size of the ICOM region depends upon the size designated by the ICOM statement. Its maximum ending address is 77756₈. This is the reason for the limitation on the size in the ICOM statement.

Next in the memory comes the area reserved for the system to store programs and the like. This area extends from the end of the ICOM region to 177617₈.

This area is followed by the registers in the read/write memory (see the list in the previous section) with a number of interspersed system-reserved areas.

Figure 6 is a graphical presentation of this organization.

The immediately addressable memory consists of 65 536 words, which is all that can be addressed by a 16-bit word (the basic unit of memory in the system). Note that the memory is divided into two blocks — an “upper” block and a “lower” one. This distinction between blocks becomes significant when addressing individual bytes in memory.

Protected Memory

All of the reserved areas mentioned above are known as “protected memory”. To give some measure of security to the operating system, it is advised that no attempt should be made to write or branch into these areas.

Access to certain portions of protected memory (e.g., BASIC variables) is provided by utilities within the assembly system. The user should access those areas only through the utilities.

Some measure of protection against access into these areas is provided during debugging. See Chapter 8 for a discussion of how this is done and the extent of the protection provided.

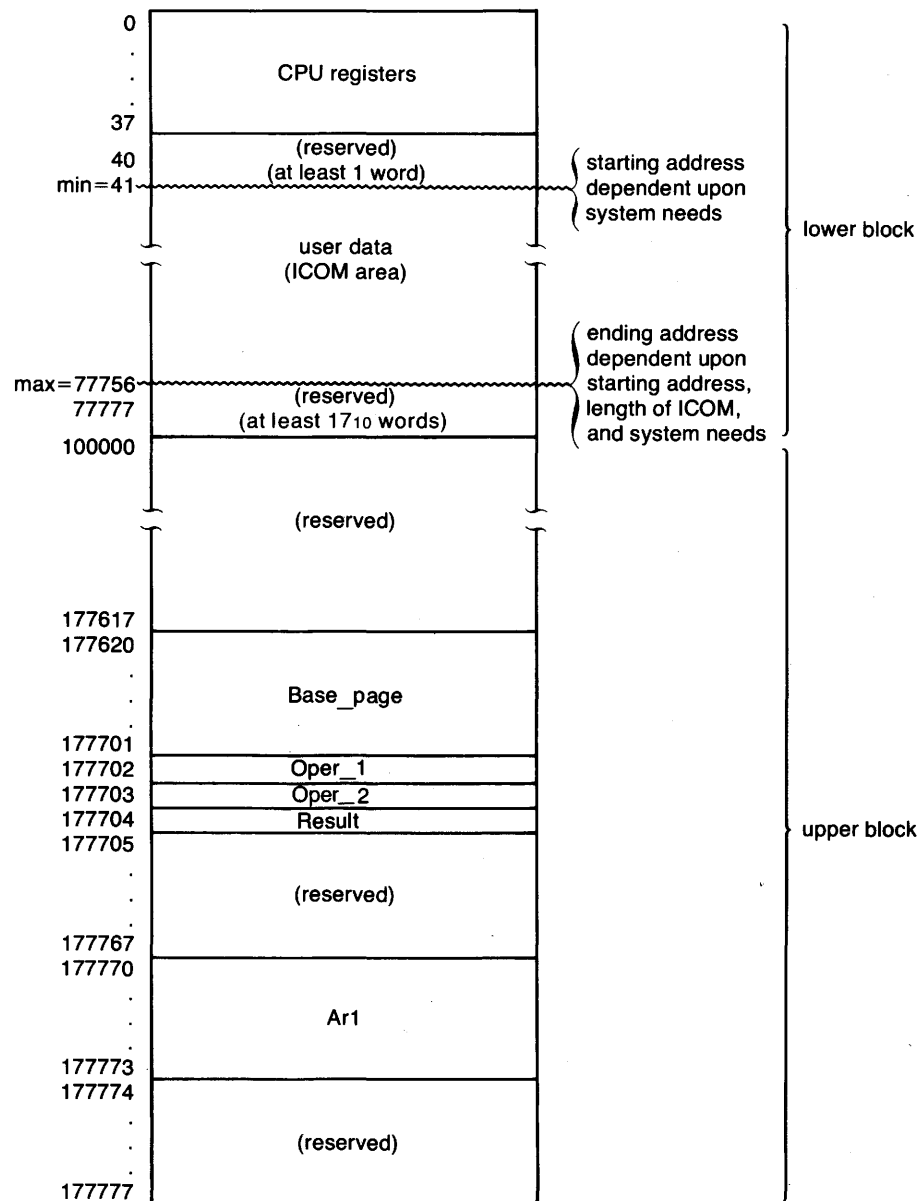


Figure 6. Memory Map

Base and Current Page

A concept that occasionally arises during discussion of the instructions and the assembler is that of the “page”, the “base” and “current” pages in particular.

A page is 1 024 words of memory.

The “base” page is a wrap-around page. It consists of the upper half of the last page in the machine (addresses 177000₈ to 177777₈) and the lower half of the zero page (addresses 0 to 777₈). This is the same as a page which runs from -512 to $+511$, effectively “wrapping around” address 0.

During execution, the program counter (P) points to the address of the current instruction. The “current” page is those 1 024 words of memory centered upon the current instruction. Therefore, the current page is a continually changing page, extending from $(P) - 512$ to $(P) + 511$.

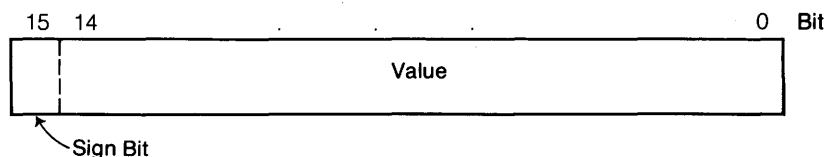
Data Structures

It is common to access BASIC variables from an assembly language routine then retrieve the contents, manipulate them, or alter them. To be effective at it, you should be aware of how BASIC stores a value in each of its data types.

There are four data types in BASIC: full-precision numeric values, short-precision numeric values, integers, and strings. Each is stored in its own unique structure.

Integers

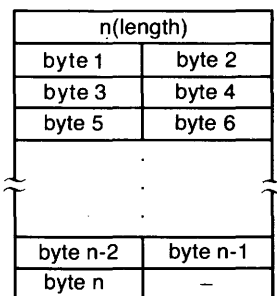
The simplest of the types is the integer. An integer consists of a single word. Values between $-32\,768$ and $+32\,767$ can be stored in the word. Negative values are stored in two’s complement form. An integer looks like —



Strings

Strings are the next simplest structure. A string is a succession of bytes, one character to a byte. A string may be of variable length. To be able to designate the length, the string is preceded by a word which contains the number of bytes in the string.

If a string has an odd number of bytes in it, then the left-over byte in the word containing the last character of the string is wasted. A typical string of length n looks like —



Full-Precision Numbers

Full-precision numeric values are stored as 12-digit, BCD (Binary Coded Decimal), floating point numbers. They occupy four words each. The first word contains the sign of the exponent, a two's-complement 10-bit exponent, and the sign of the mantissa. The other three words contain the twelve mantissa digits, 4 to each word. The words look like this —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit
Exp Sign	Exponent										0	0	0	0	0	Man Sign
D ₁ (most significant digit)				D ₂				D ₃				D ₄				
D ₅				D ₆				D ₇				D ₈				
D ₉				D ₁₀				D ₁₁				D ₁₂ (least significant)				

The exponent is always adjusted during arithmetic routines so that there is an implied decimal point following D₁. Thus, every mantissa value looks like —

$$D_1 . D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_9 D_{10} D_{11} D_{12}$$

Short-Precision Numbers

Short-precision numeric values are stored as 6-digit, BCD floating point numbers. Unlike full-precision, they occupy two words each instead of four. The first word contains a 7-bit exponent, the sign of the mantissa and the two most significant mantissa digits. The second word contains the remaining four mantissa digits. The words look like this —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit
Exp Sign		Exponent						Man Sign	D ₁				D ₂			
D ₃				D ₄				D ₅				D ₆				

As with full-precision, the exponent is stored in two's complement form and the implied decimal point follows D₁.

If you are unfamiliar with BCD arithmetic or need a refresher in floating point operations, it is suggested that you refer to Chapter 5.

Machine Instructions

The machine instruction set underlying the assembly language system consists of 92 instructions, divided into eleven groups. The groups are —

Load/Store	Operations placing values into registers or memory.
Integer Math	Operations involving integer arithmetic.
Branch	Operations altering the execution sequence unconditionally.
Test/Branch	Operations altering the execution sequence, dependent upon some condition.
Test/Alter/Branch	Operations altering the execution sequence and a value, dependent upon some condition.
Shift-Rotate	Operations performing re-arrangements of the bits in the A or B register.
Logical	Operations performing logical functions on the A or B registers.
Stack	Operations managing stacks.
BCD Math	Operations involving BCD arithmetic.
I/O	Operations specifically involving I/O operations.
Miscellaneous	Some unclassifiable operations.

Operands

Most instructions require operands. These operands have general forms which they may assume.

Many instructions contain an operand which is the address on which the function is to be performed. This {location} may be a constant (octal or decimal) or it may be a symbol. It also may be an expression containing any allowable combination of constants and symbols. For a full discussion of allowable expressions and symbols, and the “types” they are allowed to assume, consult “Symbolic Operations” in Chapter 4.

For example, note the operands in the following —

```
LDA 10B
STA Save
JMP Store + 3
AND = 177000B
```

A {location} may be either “relocatable” or “absolute” (see “Relocation” and “Symbolic Operations” in Chapter 4 for a full treatment of these types). If a relocatable {location} is used, the assembler generates machine code which uses “current page” addressing, and thus the {location} must be within — 512 words and + 511 words of the instruction. If an absolute {location} is used, the assembler generates machine code which uses “base page” addressing (meaning it takes the address as an offset from location 0).

An {address} is a {location} the same as above, except the intended location must be relocatable and within — 32 and + 31 words of the current instructions.

A {register} may be specified either through its absolute address or by its pre-defined symbol. The permissible registers are those with addresses between 0 and 7, inclusive. These are registers A, B, P, R, R4, R5, R6, and R7.

A number of instructions are followed by a {value}, which is a numeric expression usually in the range of 1 through 16. This {value} frequently indicates the number of bits involved in the operation. For example —

```
SAR 8
```

right-shifts the A register by 8 bits.

NOTE

Specifying the R4, R5, R6, or R7 registers (absolute locations 4 through 7) in an instruction causes an “I/O bus cycle” to occur. Consult Chapter 7, “I/O Handling”, for the proper use of these registers.

Indirect Addressing

Some instructions may also employ “indirect addressing”. This is indicated by including the optional indicator , I, such as —

```
LDA 10B, I
STA Save, I
JMP Store+3, I
```

There is only one level of indirect addressing provided with the processor. Of course, if further levels are desired, it is possible to implement them on your own. Some flagging scheme could be adopted, for example. One approach could be to adopt the policy that the sign bit (bit 15) of a word would indicate further indirection, with the remaining bits being the value. In such an approach, a load accumulator instruction would become two instructions —

```
10  ISOURCE  LDA A, I      ! Use current contents as pointer
20  ISOURCE  SAM *-1      ! If bit 15 set, indirection
```

Load/Store Group

This group of instructions allows transfers of data to take place. With the instructions below you can move information to and from the arithmetic accumulators (the A and B registers). You can also transfer the contents of one contiguous set of words in memory to another contiguous set.

Instruction	Description
LDA {location} [, I]	Loads register A with the contents of the specified location.
LDB {location} [, I]	Loads register B with the contents of the specified location.
STA {location} [, I]	Stores the contents of the A register into the specified location.
STB {location} [, I]	Stores the contents of the B register into the specified location.
CLR {value}	Clears (zeroes out) the specified number of words, beginning at the location specified by the A register. {value} must be an integer between 1 and 16.
XFR {value}	Transfers the specified number of words, from one location to another. The starting address of the location being transferred from must be stored in the A register. The starting address of the location being transferred to must be stored in the B register. {value} must be an integer between 1 and 16.

Integer Math Group

This group of instructions allows you to perform fundamental arithmetic operations on the contents of the arithmetic accumulators (the A and B registers).

Instruction	Description
ADA {location} [, I]	Adds the contents of the specified location to the contents of the A register, leaving the result in A. If a carry occurs, the Extend flag is set in the processor. If an overflow occurs (a carry from bits 14 or 15, but not both), the Overflow flag is set in the processor.
ADB {location} [, I]	Adds the contents of the specified location to the contents of the B register, leaving the result in B. If a carry occurs, the Extend flag is set in the processor. If an overflow occurs (a carry from bits 14 or 15, but not both), the Overflow flag is set in the processor.
TCA	Performs a two's complement of the A register (i.e., one's complement, incremented by 1). If a carry occurs, the Extend flag in the processor is set. If an overflow occurs (a carry from bits 14 or 15, but not both), the Overflow flag in the processor is set.
TCB	Performs a two's complement of the B register (i.e., one's complement, incremented by 1). If a carry occurs, the Extend flag in the processor is set. If an overflow occurs (a carry from bits 14 or 15, but not both), the Overflow flag in the processor is set.
MPY	Binary multiply. Uses Booth's Algorithm. The values of the A and B registers are multiplied together with the product placed into A and B. The A register contains the least significant bits and the B register contains the most significant bits and the sign. (An anomaly in the processor results in an improper result whenever A or B equals - 32 768.)

Branch Group

This group of instructions allows you to alter the execution sequence unconditionally. It includes the “jumps” and “returns” from subroutines.

Instruction	Description
JMP {location} [, I]	Unconditionally branches to the specified location.
JSM {location} [, I]	Jumps to a subroutine. The value of the R register is incremented and the current value of the P register (i.e., the location of the JSM instruction itself) is stored into the address pointed to by the R register. Execution then proceeds to the specified location.
RET {value}	Returns from a subroutine. {value} is added to the contents of the address pointed to by the R register. The results are stored in the P register (i.e., specifying the next location for execution) and the R register is decremented. This is, in effect, a return from a JSM instruction to the instruction which is {value} instructions from the JSM itself. The “usual” return is RET 1. {value} must be an integer between -32 and 31.

Test/Branch Group

Similar to the Branch group, this group of instructions allows you to alter the execution sequence, but conditionally upon the result of some test. Most instructions involve tests on all or part of one of the arithmetic accumulators (the A and B registers), but a couple allow a test on a location in memory which you can specify.

Instruction	Description
CPA {location} [, I]	Compares the contents of the A register with the contents of the specified location. Execution skips over the next word if the contents are not equal.
CPB {location} [, I]	Compares the contents of the B register with the contents of the specified location. Execution skips over the next word if the contents are unequal.
SZA {address}	Skips to {address} if register A is 0.
SZB {address}	Skips to {address} if register B is 0.
RZA {address}	Skips to {address} if register A is not 0.
RZB {address}	Skips to {address} if register B is not 0.
SIA {address}	Skips to {address} if register A is 0, then increments A regardless. The Extend and Overflow flags in the processor are not affected by the incrementing action.
SIB {address}	Skips to {address} if register B is 0, then increments B regardless. The Extend and Overflow flags in the processor are not affected by the incrementing action.
RIA {address}	Skips to {address} if register A is not 0, then increments A regardless. The Extend and Overflow flags in the processor are not affected by the incrementing action.
RIB {address}	Skips to {address} if register B is not 0, then increments B regardless. The Extend and Overflow flags in the processor are not affected by the incrementing action.

Test/Alter/Branch Group

Similar to the Test/Branch group, this group of instructions allows you to conditionally alter the execution sequence. In addition to tests, you can also alter the contents of the item being tested (such as set or clear a bit, or increment or decrement a register). Certain bits in the processor (Extend and Overflow) can be tested with some of these instructions, as well as registers and memory locations.

Some instructions may be followed by either of the following —

, S
, C

indicating that the bit being tested by the instruction will either be set (S) or cleared (C) after the test has been made.

Instruction	Description
ISZ {location} [, I]	Increment the contents of the specified location and skip execution of the next word if the result is 0.
DSZ {location} [, I]	Decrement the contents of the specified location and skip execution of the next word if the result is 0.
SAP {address} [, S] SAP {address} [, C]	Skips to {address} if the A register is positive or zero (bit 15 is 0).
SBP {address} [, S] SBP {address} [, C]	Skips to {address} if the B register is positive or zero (bit 15 is 0).
SAM {address} [, S] SAM {address} [, C]	Skips to {address} if the A register is negative (bit 15 is 1).
SBM {address} [, S] SBM {address} [, C]	Skips to {address} if the B register is negative (bit 15 is 1).
SLA {address} [, S] SLA {address} [, C]	Skips to {address} if the least significant bit of the A register is 0.

Instruction	Description
SLB {address} [, S]	Skips to {address} if the least significant bit of the B register is 0.
SLB {address} [, C]	
RLA {address} [, S]	Skips to {address} if the least significant bit of the A register is not 0.
RLA {address} [, C]	
RLB {address} [, S]	Skips to {address} if the least significant bit of the B register is not 0.
RLB {address} [, C]	
SOS {address} [, S]	Skips to {address} if the Overflow flag in the processor is set.
SOS {address} [, C]	
SOC {address} [, S]	Skips to {address} if the Overflow flag in the processor is cleared.
SOC {address} [, C]	
SES {address} [, S]	Skips to {address} if the Extend flag in the processor is set.
SES {address} [, C]	
SEC {address} [, S]	Skips to {address} if the Extend flag in the processor is cleared.
SEC {address} [, C]	

NOTE

The Extend and Overflow flags can be cleared only by using the SEC, SES, SOC, and SOS instructions with the , C option.

Shift/Rotate Group

This group of instructions performs re-arrangements of bits in the arithmetic accumulators (the A and B registers). Circular and non-circular shifts are available.

Instruction	Description
SAR {value}	Shifts the A register right the indicated number of bits with all vacated bit positions becoming 0.
SBR {value}	Shifts the B register right the indicated number of bits with all vacated bit positions becoming 0.
SAL {value}	Shifts the A register left the indicated number of bits with all vacated bit positions becoming 0.
SBL {value}	Shifts the B register left the indicated number of bits with all vacated bit positions becoming 0.
AAR {value}	Shifts the A register right the indicated number of bits with the sign bit filling all vacated bit positions. (Arithmetic right)
ABR {value}	Shifts the B register right the indicated number of bits with the sign bit filling all vacated positions. (Arithmetic right)
RAR {value}	Rotates the A register right the indicated number of bits. Bit 0 rotates into bit 15 each time. (Right circular)
RBR {value}	Rotates the B register right the indicated number of bits. Bit 0 rotates into bit 15 each time. (Right circular)
RAL {value}	Rotates the A register left the indicated number of bits. Bit 15 rotates into bit 0 each time. (Left circular)
RBL {value}	Rotates the B register left the indicated number of bits. Bit 15 rotates into bit 0 each time. (Left circular)

Logical Group

This group of instructions performs logical (Boolean) operations upon the contents of an arithmetic accumulator (on A or B register). Logical “and” and “or” operations are available, along with complementing and clearing operations.

Instruction	Description
AND {address} [, I]	Logical “and” operation. The contents of the A register are compared bit by bit, with the contents of the specified location. For each bit-comparison a 1 results if both bits are 1’s, a 0 results otherwise. The 16-bit result is left in A.
IOR {address} [, I]	Logical “inclusive or” operation. The contents of the A register are compared, bit by bit, with the contents of the specified location. For each bit-comparison, a 0 results if both bits are 0’s, a 1 otherwise. The 16-bit result is left in A.
CMA	Performs a one’s complement of the A register (i.e., bit-by-bit inversion of all 16 bits).
CMB	Performs a one’s complement of the B register (i.e., bit-by-bit inversion of all 16 bits).
CLA	Clears register A. This instruction is identical to SAR 16.
CLB	Clears register B. This instruction is identical to SBR 16.

Stack Group

The Stack group of instructions provides you with operations for managing stacks. The instructions withdraw items from (also called “pop” or “pull”) or push items onto a stack pointed to by either the C or D register. The items are pushed from or withdrawn into a specified register (other than C or D) and the C or D register is incremented or decremented appropriately.

Pushing instructions increment or decrement the C or D register prior to doing the pushing. Withdrawing instructions increment or decrement the C or D register after doing the withdrawal. Consequently, the pointer is always left pointing to the “top” of the stack after the operation.

Decrementing the C or D register is indicated by including , D after the operand. For “withdrawing” instructions, D is the default. For example, the following are equivalent —

```
WWC A, D
WWC A
```

Incrementing is specified by including , I after the operand. This is also the default for “pushing” instructions if neither I or D is included. For example, the following are equivalent —

```
PWC A, I
PWC A
```

When using the byte instructions (PBC, PBD, WBC, WBD), the address pointed to by the C or D register must not have an absolute address less than 40s.

When pushing or withdrawing bytes, the least significant bit of the address register (either C or D) is used to determine which byte is desired in the stack (a 0 implies the left most byte of the word being addressed). To retain the full 16-bit addressing capability, the Cb or Db register is used, as appropriate. These one-bit registers hold the most significant bit of the word address when the byte addressing instructions are used. They should be explicitly set or cleared, depending upon the value of the address involved.

Instruction	Description
PWC {register} , D PWC {register} [, I]	Pushes contents of {register} onto the stack pointed to by the C register.
PWD {register} , D PWD {register} [, I]	Pushes contents of {register} onto the stack pointed to by the D register.
PBC {register} , D PBC {register} [, I]	Pushes the lower byte (right half) of {register} onto the stack pointed to by the Cb and C registers. If the least significant bit of C is a 1, the byte is placed in the lower byte of the word in the stack; if it is a 0, it is pushed into the upper byte.
PBD {register} , D PBD {register} [, I]	Pushes the lower byte (right half) of {register} onto the stack pointed to by the Db and D registers. If the least significant bit of D is a 1, the byte is placed in the lower byte of the word in the stack; if it is a 0, it is pushed into the upper byte.
WWC {register} [, D] WWC {register} , I	Withdraws a word from the stack pointed to by the C register and stores it into {register}.
WWD {register} [, D] WWD {register} , I	Withdraws a word from the stack pointed to by the D register and stores it into {register}.
WBC {register} [, D] WBC {register} , I	Withdraws a byte from the stack pointed to by the Cb and C registers and places it into the lower byte (right half) of {register}. If the least significant bit of C is a 1, the byte is withdrawn from the lower byte of the word in the stack; if it is a 0, it will be withdrawn from the upper byte.
WBD {register} [, D] WBD {register} , I	Withdraws a byte from a stack pointed to by the Db and D registers and places it into the lower byte (right half) of {register}. If the least significant bit of D is a 1, the byte is withdrawn from the lower byte of the word in the stack; if it is a 0, it is withdrawn from the upper byte.
CBL	Clears the Cb register (indicates lower block of memory).
CBU	Sets the Cb register (indicates upper block of memory).
DBL	Clears the Db register (indicates lower block of memory).
DBU	Sets the Db register (indicates upper block of memory).

BCD Math Group

This group of instructions provides you with BCD arithmetic operations using the Ar1 and Ar2 registers.

In general, the instructions associate the Ar1 register with “X” and the Ar2 register with “Y” in the mnemonic for the instruction. Both registers contain values which are considered BCD full-precision values when operated upon by instructions in this group.

The mantissas referred to below consist of 12 BCD digits. All the shifting operations manipulate the digits as units (i.e., 1 digit — or 4 bits — at a time). In addition, shifting operations involve an additional digit in the A register (located in the lower 4 bits, numbered 0 through 3).

All arithmetic is performed in BCD. The values being operated upon are assumed to be normalized BCD floating-point (full-precision) values. Signs and exponents are left strictly alone. There is a flag in the processor, called Decimal Carry, which is set when an overflow occurs during a BCD operation.

A full discussion of BCD arithmetic techniques can be found in Chapter 5.

Instruction	Description
MRX	<p>Mantissa right shift on Ar1. The number of digits to be shifted is specified in the lower 4 bits (0-3) of the B register. The shift is accomplished in three stages —</p> <ol style="list-style-type: none"> <li data-bbox="581 1310 1334 1436">1. The digit in bits (0-3) of the A register is right-shifted into the first digit of the mantissa, with the twelfth digit being lost. This is the first shift. <li data-bbox="581 1457 1334 1625">2. The mantissa digits are then right-shifted for the remaining number of digits specified. The twelfth digit, except for the last shift, is lost on each shift and the vacated digits are zero-filled. <li data-bbox="581 1646 1334 1814">3. Finally, the last right-shift takes place with the twelfth digit shifting into the A register. The Decimal Carry flag in the processor is cleared along with the upper 12 bits of the A register (4-15).

Instruction	Description
MRY	<p>Mantissa right-shift on Ar2. The number of digits to be shifted is specified in the lower four bits (0-3) of the B register. The shift is accomplished in three stages —</p> <ol style="list-style-type: none"> 1. The digit in bits (0-3) of the A register is right-shifted into the first digit of the mantissa, with the twelfth digit being lost. This is the first shift. 2. The mantissa digits are then right-shifted for the remaining number of digits specified. The twelfth digit, except for the last shift, is lost on each shift, and the vacated digits are zero-filled. 3. Finally, the last right-shift takes place, with the twelfth digit shifting into the A register. The Decimal Carry flag in the processor is cleared along with the upper 12 bits of the A register (4-15).
MLY	<p>Mantissa left-shift on Ar2 for one digit. This is a circular shift, with the digit in bits (0-3) of the A register forming a thirteenth digit. The non-digit part of the A register is cleared (i.e., bits 4-15), and the Decimal Carry flag in the processor is cleared.</p>
DRS	<p>Mantissa right-shift on Ar1 for one digit. The twelfth digit is shifted into the A register (bits 0-3). The non-digit part of the A register is cleared (i.e., bits 4-15), and the Decimal Carry flag in the processor is cleared. The first digit in the mantissa is set to 0.</p>
NRM	<p>Normalizes the Ar2 mantissa. The mantissa digits are left-shifted until the first digit of the mantissa is non-zero, or until twelve shifts have taken place, whichever comes first. If the original first digit is already non-zero, no shifts occur. The number of shifts required is stored as the first four bits (0-3) of the B register. If twelve shifts were required, the Decimal Carry flag in the processor is set, otherwise it is cleared.</p>
CMX	<p>Ten's complement of Ar1. The mantissa of Ar1 is replaced with its ten's complement and Decimal Carry is cleared.</p>

Instruction	Description
CMY	Ten's complement of Ar2. The mantissa of Ar2 is replaced with its ten's complement and Decimal Carry is cleared.
FXA	Fixed-point addition. The mantissas of Ar1 and Ar2 are added together, and the result is placed into Ar2. Decimal Carry is added to the twelfth digit. After the addition, Decimal Carry is set if an overflow occurred, otherwise Decimal Carry is cleared.
MWA	Mantissa word addition. The contents of the B register are added to the ninth through twelfth digits of the mantissa of Ar2. Decimal Carry is added to the twelfth digit; if an overflow occurs, Decimal Carry is set, otherwise it is cleared.
FMP	Fast Multiply. Performs the multiplication by repeated additions. The mantissa of Ar1 is added to the mantissa of Ar2 a specified number of times. The number of times is specified in the lower 4 bits (0-3) of the B register. The result accumulates in Ar2. If intermediate overflows occur, the number of times they occur appears in the lower 4 bits of the A register after the operation is complete. The upper 12 bits of the A register are cleared along with Decimal Carry.
FDV	Fast divide. The mantissas of Ar1 and Ar2 are added together until the first decimal overflow occurs. The result accumulates into Ar2. The number of additions without overflow is placed into the lower 4 digits of the B register (0-3). The remainder of the B register is cleared, as is the Decimal Carry flag in the processor.
CDC	Clears the Decimal Carry flag in the processor.
SDS {address}	Skips to {address} if Decimal Carry is set. Decimal Carry is a flag in the processor which may be set as the result of certain BCD arithmetic operations (see Chapter 5 for details).
SDC {address}	Skip to {address} if Decimal Carry is cleared. Decimal Carry is a flag in the processor which may be set as the result of certain BCD arithmetic operations (see Chapter 5 for details).

I/O Group

The I/O group of instructions provides you with some of the operations necessary to accessing peripheral devices through the I/O bus. In addition to the instructions contained here, there are instructions in other groups which can have I/O effects (e.g., LDA, STA...).

The techniques useful to the implementation of I/O operations using the instructions in this group and the other groups are discussed in Chapter 7.

Instruction	Description
SFS {address}	Skips to {address} if the Flag line is set (true). The Flag line is associated with a peripheral on the current select code (see Chapter 7 for details).
SFC {address}	Skips to {address} if the Flag line is clear (false). The Flag line is associated with a peripheral on the current select code (see Chapter 7 for details).
SSS {address}	Skips to {address} if the Status line is set (true). The Status line is associated with a peripheral on the current select code (see Chapter 7 for details).
SSC {address}	Skips to {address} if the Status line is clear (false). The Status flag is associated with a peripheral on the current select code (see Chapter 7 for details).
EIR	Enables the interrupt system. Cancels the DIR instruction.
DIR	Disables the interrupt system. Cancels the EIR instruction.
SDO	Sets DMA outwards. Directs that DMA operations read from memory, write to the peripheral.
SDI	Sets DMA inwards. Directs that DMA operations read from the peripheral, write to memory.
DMA	Enables the DMA mode. Cancels the DDR instruction.
DDR	Disables Data Request. Cancels the DMA instruction.

Miscellaneous

The following instructions are unclassifiable into any of the other groups.

Instruction	Description
NOP	Null operation. This is exactly equivalent to LDA A.
EXE {value} [, I]	The contents of any register can be treated as the current instruction and executed. {value} is a numeric expression in the range 0 through 31, indicating the register to be used. The register is left unchanged, unless the instruction code causes it to be altered. The next instruction to be executed is the one in the word following the EXE, unless the code in the executed register causes a branch.

Chapter 4

Table of Contents

Assembly Language Fundamentals

Program Entry	49
Assembly Language Source	51
Actions	51
Labels	51
Comments	53
Syntaxing the Source	53
Creating Modules	55
Storage	56
Modules	56
Variables	56
Data Generators	57
Repeating Instructions	59
Assembling	60
Effect of BASIC Environments	60
Source Listing Control	61
Page Format	62
Page Length	63
End-of-Page Control	63
Page Headings	64
Blank Line Generation	65
Non-Listable Pseudo-Instructions	65
Conditional Assembly	65
Relocation	68
Symbolic Operations	69
Pre-Defined Symbols	69
Defining Your Own	71
Literals	72
Evaluation of Literals	72
Nesting Literals	73
Nonsensical Uses of Literals	74
Literal Pools	74
Expressions	75
External Symbols and Elements	77
Other Absolute Elements	78
Utilities	79

Chapter 4

Assembly Language Fundamentals

Summary: This chapter discusses some of the basic statements and syntaxes used throughout the assembly language system. Program entry, assembling, symbolic operations, module creation, program and variable storage, and utilities are the topics covered.

When writing assembly language programs there are a number of things with which you will be involved constantly. In the beginning, questions arise on how to use the language: How do you enter the source code? What kind of symbolic addressing is there? How do you create and distinguish modules? How do you create the object code and where is it stored? What utilities are available and how do you use them?

The answers to those questions form the underlying capabilities through which you write your applications. These are things which nearly every assembly language program uses. As essential as they are, however, none are difficult to master.

Program Entry

You were introduced early in Chapter 2 to the integrated nature of the assembly language with its host language, BASIC. You know from that chapter how assembly language statements can be intermingled with BASIC statements — that you can employ the usual editing features on the assembly statements. However, there is more to the ISOURCE statement than just its integrated nature with BASIC.

As stated in Chapter 2, all assembly language statements are designated with the keyword “ISOURCE”. The keyword is followed by {assembly language source}. So the syntax of the entry line is —

```
{line number} [ {BASIC label} : ] ISOURCE {assembly language source}
```

Here's a simple example of this from Chapter 2 —

```
40  ISOURCE  NAM Example
50  ISOURCE  NOP
60  ISOURCE  END Example
```

The {line number} and {BASIC label} are the same as you are used to in BASIC. However, it should be noted that the statement is not an executable one, so the BASIC label is only useful for documentation and EDIT purposes.

To BASIC, the ISOURCE statement appears as a comment. If you were to change the above so that it read —

```
40 Example:  ISOURCE  NAM Example
50           ISOURCE  NOP
60           ISOURCE  END Example
70           END
```

and then executed a statement “GOTO Example”, the result would be to simply execute the END statement in line 70. That is because, to BASIC, the lines appear the same as —

```
40 Example:  REM
50           REM
60           REM
70           END
```

or —

```
40 Example:  !
50           !
60           !
70           END
```

The BASIC label on an ISOURCE line finds its most useful characteristic in being able to be referenced, as any other BASIC label on any other type of line may be, with an EDIT command. Thus, if you were to execute —

```
EDIT Example
```

on the above, you would be working in the editor, starting with line 40. This feature will become useful during program development as will be pointed out shortly.

Assembly Language Source

You may have recognized the assembly language instruction and pseudo-instructions to the right of ISOURCE in the examples above. This is where your instructions and pseudo-instructions appear. However, the source is a little more versatile than that. In general, {assembly language source} has the syntax —

```
[ {label} : ] {action} [ ! {comment} ]
```

Or, the action may be omitted and only a comment appears —

```
[ {label} : ] ! {comment}
```

A label is always optional in the source, but either an {action} or a {comment} must be present in every source line.

Actions

An {action} in assembly language source is —

- A machine instruction, with any operand it may require. These were discussed at some length in Chapter 3.
- A pseudo-instruction, with any operand it may require. These are discussed under the topics to which they relate.

The actions contained in the above example were —

```
NAM Example
NOP
END Example
```

Labels

The {label} in assembly language source is part of the symbolic addressing capability of the assembler. This {label} is used by the **assembler** only. Neither the operating system nor BASIC is aware of its existence.

The label follows the same form and rules as do labels in BASIC —

- Up to 15 characters long.
- First character must be a capital letter (A-Z).
- Only the non-capital letters (a-z), the numerals (0 to 9), or the underscore (_) may be used following the first character.

No two labels are allowed to be the same in a given **module**. If your source consists of two or more modules, then the same label may be defined more than once, provided each definition is in a different module. (Distinguishing between modules is discussed in “Creating Modules”, later in this chapter.) So you may not code —

```
Rumpelstiltskin: LDA B
```

in one place in the module and later in the same module code —

```
Rumpelstiltskin: LDB A
```

There are other restrictions as well on the choosing of labels. For instance, there are symbols already defined by the assembler and you are not allowed to choose one of them as a label. This is discussed at length in “Symbolic Operations” in this chapter.

Both a BASIC label AND an assembly language source label can appear in the same line, and they are distinct from one another. BASIC does not know about the source label and the assembly language system does not know about the BASIC label.

Since neither BASIC nor the operating system is aware of the existence of source labels, actions outside the assembler cannot reference these labels. Thus, if you had the source line —

```
100  ISOURCE Rumpelstiltskin: JMP Bail
```

You can neither say GOTO Rumpelstiltskin nor EDIT Rumpelstiltskin. Neither of these can find “Rumpelstiltskin”, since only the assembler can know it is there.

This can be a nuisance in some instances during program development. Many programmers use labels almost exclusively and rarely consider the line number when using the editor to change a line. For instance, in the above, they would not be used to saying, “EDIT 100” to get at the line in order to change it. They are more used to saying, “EDIT Rumpelstiltskin”. A way for them to do it would be to change the line to —


```
100 Rumpelstiltskin: ISOURCE Rumpelstiltskin: JMP Bail
```

Note that, as the example demonstrates, the name can be the same in the BASIC label as in the source. This takes advantage of the fact that BASIC and the assembler are unaware of each other’s labels. The names do not have to be the same.

Comments

As with any BASIC line, a comment may be included by simply adding an exclamation point (!) and typing your comment after it. Since you have a total of 160 characters for a line, your comment may fill up the remainder of the 160 characters left after the rest of the statement has been provided (line number, ISOURCE keyword, label, action).

Syntaxing the Source

When you are creating your source program, you are either entering it from the keyboard or retrieving it from mass storage (LINK or GET). In either case, as the statement is entered (the  key on the keyboard is pressed or a record is read from mass storage), the operating system takes note of any use of the keyword ISOURCE. When a line has this keyword, the operating system turns over the remainder of the line following the keyword to the assembly system. The assembly system, then and there, checks the syntax of the source.

By checking the syntax at the time of entry of the statement, a considerable amount of processing time is saved when the time comes to assemble the source into object code. In addition, it gives you, as the programmer, immediate feedback when a syntactical error occurs. You do not have to wait until assembly time just to find out that you misspelled NOP.

At syntax time, the assembler takes care of capitalization, lower case, and spacing for the source. It's quite similar to the SPACE DEPENDENT mode of entry for BASIC statements (that mode is not required to get the effect with the assembly system). It follows the following rules in syntaxing the source —

- Everything between the ISOURCE and the colon (if present) is the label. Its initial character is capitalized and the remaining letters are converted to lower-case. This is regardless of whether they were entered in that form.
- The label, if present, is left-justified to the second column following the keyword ISOURCE.
- The first three letters following the colon (or just the first three letters, if there is no label) are considered the machine instruction or pseudo-instruction and are capitalized. The instruction will remain in the same column as it was entered, and, if possible, a space is added after it.
- Everything after the instruction or pseudo-instruction is considered the operand for the instruction, up until the exclamation point before the comment (if any). Any label (symbol) in the operand will have its initial character capitalized and the remaining letters converted to lower case automatically.
- Comments are unchanged and remain in the same columns as entered, whenever possible.

In short, simply enter the statement in your most comfortable fashion and the assembly system automatically assures that what you enter is in the proper form (though it still can't guarantee that you have entered the right instruction for what you mean to do).

As a demonstration of this facility, consider the following line ready for syntaxing —

```
100 ISOURCE      RUMPELSTILTSKIN:jMpbail
```

It becomes —

```
100 ISOURCE Rumpelstiltskin:    JMP Bail
```

Creating Modules

When you were introduced in Chapter 2 to the concept of a module, it was said that a module is given a name through the NAM pseudo-instruction.

So, when you enter a source line which has the following form —

```
NAM {module name}
```

you are assigning a name to a module, and you are also delimiting the beginning of the module. By the inclusion of this statement, all source lines which follow are part of the module with the name designated in this source line, that is, all lines until the END pseudo-instruction is encountered in the source. It has the form —

```
END {module name}
```

Its {module} name must be the same as in the NAM pseudo-instruction.

A {module name} follows the same rules for naming as do labels (see above).

It is by the use of these two instructions that modules are created. The source lines which appear between them comprise a single module, and the name assigned to the module is the one with which the module is referenced (with the ILOAD and ISTORE statement for example).

When it comes time to assemble the source into object code, the assembler treats the source lines in a module as a unit.

In actuality, therefore, there are **two** modules — a source module and an object module. When you are assembling a module, the name you use refers to the source module and creates the object module. Later, other statements, such as ISTORE and ILOAD, refer solely to the object module.

Storage

Modules

When assembly converts a source module into an object module, there must be a place to keep the object module. That is the function of the ICOM region.

You were introduced to the ICOM region in Chapter 2 in connection with the loading and storing of modules. It is also used to hold modules which are created through assembly. Once a module has been assembled, the object code appears in the ICOM region just as if you had loaded it from mass storage.

Variables

Within a module, you may want to set aside one or more words of memory for your use. For example, you might need a location to store a variable, or keep a counter, or save a register. This is done with the BSS pseudo-instruction —

```
BSS {number}
```

where {number} is the number of words to be set aside. {number} can be any absolute expression, provided the expression evaluates to a positive integer (see “Symbolic Operations” below).

This kind of storage is part of the object code and is set aside “in-line”. This means that wherever it appears in the source, the storage appears in the same relative location in the object module.

For example, suppose a module contained the following source lines —

```

220  ISOURCE Save_a:  BSS 1
230  ISOURCE Save_4:  BSS 2*2
240  ISOURCE Renras:  BSS Larry
250  ISOURCE Again:   LDA Renras

```


Then, at some appropriate spot in the object module (relative to the other instructions in the module) there would be the following **contiguous** locations —

```
Save_a  1 word
Save_4  4 words
Renras  some number of words equal to "the absolute symbol, Larry"1
Again   1 word
```

The locations at labels `Save_a`, `Save_4`, and `Renras` are merely reserved by the BSS pseudo-instructions, and their contents are not initialized to any particular value.

It is possible to accidentally execute these locations when the routine is run if you're not careful. Ordinarily, you should place these locations somewhere safely out of the potential execution sequence, since they are used just for storage. Some applications, though, use self-generating code, and a BSS is a way to set aside locations for it.

Data Generators

A "data generator" is very much like a BSS operation. The function, as with the BSS, is to set aside words of memory at a particular location in the object code. But in addition, the words are to be initialized to some value. The initialization occurs at the same time the words are set aside (i.e., at assemble-time).

This is done using the DAT pseudo-instruction which has the form —

```
DAT {expression} [, {expression} [, ...]]
```

An {expression} may be any absolute or relocatable expression. The various forms that an expression may take are discussed in "Symbolic Operations" later in this chapter.

As an example, suppose you want the value 100 (a decimal integer) to be located at location "X" in the object module. You can achieve this by identifying the location in the source code (ultimately the object code) where you want the value to be, then placing this instruction at that point —

```
X: DAT 100
```

¹ Such symbols are discussed at length in the "Symbolic Operations" section later in this chapter.

Upon encountering this pseudo-instruction, the assembler generates the words necessary to store the value (in this case, only 1 word is necessary). It then stores the value (100) into the word(s) and proceeds with the remaining assembly. Thus, the location of the words is dependent upon the instruction's relative position in the source module, the same as with any machine instruction.

The number of data words generated for each {expression} is dependent upon the result of the {expression} —

Result	Words
Full-precision	4
Short-precision	2
Decimal integer	1
Octal integer	1
Address ¹	1
Literal	1
String	actual length (2 characters per word)

If more than one {expression} is present, the necessary data words are generated in the order in which they appear in the list. As an example, if you were to include the instruction —

```
DAT 2,2S,2.0,"2",/2,=2
```

ten words would be set aside and initialized to the appropriate values —

```
000002 -2
000040 } 2S
000000 }
000000 }
020000 } 2.0
000000 }
000000 }
031000 -"2"
000062 -'2'
000053 - address of 2 in literal pool
```

¹ including "external"

Repeating Instructions

To help relieve the tedium of writing the same instruction many times (which many applications occasionally require), a “repeat” pseudo-instruction is provided —

```
REF {expression}
```

The pseudo-instruction causes the immediately following machine instruction to be duplicated in the object code {expression} number of times.

For example, suppose you are writing a real-time application where timing was critical, and to make things work correctly you need 10 NOPs at a certain location. Ordinarily you would type —

```
ISOURCE  NOP
ISOURCE  NOP
ISOURCE  NOP
ISOURCE  NOP
ISOURCE  NOP
ISOURCE  NOP
ISOURCE  NOP
ISOURCE  NOP
ISOURCE  NOP
ISOURCE  NOP
```

But all of this could be replaced with —

```
10  ISOURCE  REP 10
20  ISOURCE  NOP
```

and the same effect would be achieved.

Some pseudo-instructions may not be replicated. They are —

```
COM
END
EQU
EXT
NAM
REP
SUB
```

Assembling

Object code is created by “assembling” the source code. Again, modules are a key factor. The assembly directive is aimed at modules, using the module name as a delimiter in the source code so the assembler can tell which ISOURCE statements to assemble as part of the module. Of course this same name is also used to store the object code using mass storage.

The IASSEMBLE statement is the vehicle for assembling modules. It has the forms —

```
IASSEMBLE {module} [ , {module} [ , ... ] [ ; {option} [ , {option} [ , ... ] ] ]
IASSEMBLE [ ALL ] [ ; {option} [ , {option} [ , ... ] ] ]
```

Each {module} indicated is assembled, in the order given by the statement. Only those modules are assembled; any others which may be present in the source at the time are ignored. If the ALL version of the statement is used (with or without the optional word ALL), every module present in the source is assembled.

An {option} falls into one of two categories: listing directives and conditions (for conditional assembly). These are discussed separately below. The options, and their categories, are —

EJECT	}	Listing directives
LINES		
LIST		
XREF		
A	}	Conditions
B		
C		
D		
E		
F		
G		
H		

Effect of BASIC Environments

To assemble a module, all of its source lines (between the NAM and END pseudo-instructions) must lie within the same BASIC “environment”. That is, the NAM and END for a module must lie within the main program or within the same subprogram or multi-line function. For modules where this is not true, an error (“EN” assemble-time error) occurs.

Source Listing Control

Listings of the source code in a module can be obtained during an assembly. These listings contain the line numbers, instructions, and comments from the source lines along with the associated machine addresses and contents of that address.

Here is part of a typical listing —

```

430 01034 002645 LDA =Array_type
440 01035 006645 LDB =Array
450 01036 142645 JSM Get_info      !Info on the array
460 01037 003005 LDA Array_type    !Look at the type
470 01040 012644 CPA =16        !Is it a file number?
480 01041 066003 JMP **3          !Must be a file number
490 01042 022643 ADA =-12       !Is it an array data
500 01043 172003 SAP **3        ! type (ie, >12)?
  
```

line numbers absolute addresses contents actions comments

The addresses and contents are displayed in octal representation.

Listings are not automatic. They are obtained in one of two ways —

- By using the LIST option in the IASSEMBLE statement. This directs that a listing is desired for all the modules in the statement. The statement would look like the following examples —

```

IASSEMBLE Store;LIST
IASSEMBLE Retrieve;Work;LIST
  
```

- By using the LST pseudo-instruction in the source code itself.

Modules can be just partially listed, if desired. This kind of control is achieved by using the LST and UNL pseudo-instructions within the source code, placing the LST before any instructions which you want listed, and placing the UNL before any instructions you do not want listed. For example, if the following source lines are assembled —

```

420 ISOURCE LST
430 ISOURCE LDA =Array_type
440 ISOURCE LDB =Array
450 ISOURCE JSM Get_info      !Info on the array
460 ISOURCE LDA Array_type    !Look at the type
470 ISOURCE CPA =16          !Is it a file number?
480 ISOURCE JMP **3          !Must be a file number
490 ISOURCE ADA =-12         !Is it an array data
500 ISOURCE SAP **3          ! type (ie, >12)?
510 ISOURCE UNL
  
```

only lines 430 through 500 would be listed.

The primary purpose of this capability is to allow as much modularity in the listings as you can get in source code. To implement this purpose, a “listing counter” is used.

Whenever an LST instruction is encountered during an assembly, the listing counter is incremented. Whenever an UNL instruction is encountered during an assembly, the listing counter is decremented. Source lines are listed whenever the counter is greater than 0. Whenever it is equal to 0 or negative, then no lines are listed.

The counter is set to 0 upon execution of the IASSEMBLE statement. This is why there is no automatic listing. However, if the LIST option is included in the IASSEMBLE statement, then the counter is initialized to 1. This is why that option creates a listing. Thus, you could defeat a LIST option by placing an UNL instruction at the beginning of a module. This initialization occurs for each module assembled, so if you have more than one module indicated in your IASSEMBLE statement, the counter is set at the beginning of the assembly for each.

This capability sees its greatest usefulness during debugging stages and while working with independently written sections of source code. For example, a number of people could be writing different sections of code, each containing their own LST and UNL instructions. These instructions could then be overridden when they were combined into a single module by preceding the sections with an LST instruction (to get a listing) or an UNL (to suppress the listings).

Page Format

Each and every assembly listing page has the following format —

- The word “PAGE” and the current page number of the listing occurs on the first line starting at column 49.
- A heading occurs on the second line, left-justified. The heading always includes —

MODULE: {name}

where {name} is the name of the module currently being assembled. Additional heading information can be specified for this line (see “Page Heading” below).

- A blank line follows the heading.
- The text follows the blank line. The number of lines printed depends upon the `LINES` option in the `IASSEMBLE` statement, the number of source lines encountered, and the `SKP` pseudo-instructions which may be encountered while assembling the source. `LINES` and `SKP` are described in the following sections.
- If the `EJECT` option is **not** included in the `IASSEMBLE` statement, then a minimum of three blank lines (carriage return/line feed, `CR/LF`, pairs) will be printed at the end of a page. The number may exceed three if the number of source lines printed on a page is less than the standard length for a listing page (see above).

Page Length

The length of the text in each page of your assembly listings can be specified through the `IASSEMBLE` statement using the `LINES` option, which has the form —

```
LINES {numeric expression}
```

This option directs that any listing of the routines being assembled have pages of the length indicated by {numeric expression}, which must be a positive value. This value becomes the “standard length” of the listing pages, specifying the number of source lines to be printed on a page during listings of the assembly source. It is not necessary that this value be the page length of the printing device being used, though this is frequently the value selected.

If the option is omitted from the `IASSEMBLE` statement, the value of 60 is assumed for page length, giving an overall page size of 66 lines.

Printer control characters, such as line-feed and form-feed, in a comment can affect the actual printing length of the pages independent of the length you specify. Thus, a page length of 60 could result in actually 61 lines if one of the comments in your `ISOURCE` statements contains a line-feed character.

End-of-Page Control

At any time during the assembly of a module, you can force the listing to continue printing at the top of the next physical page by including —

```
SKP
```

at the desired spot in the module. If a listing is being generated when this pseudo-instruction is encountered in the source code, the printer is sent to top-of-form. This is physically done in one of two ways —

- If the EJECT option was included in the IASSEMBLE statement which is assembling the module, then a form-feed character (ASCII character 14_s), is sent to the printer.
- If the EJECT option was not included, sufficient CR/LF pairs (ASCII characters 15_s and 12_s) are sent to the printer to fill out the standard length of a listing page (plus three at the end of the page). Thus, if you already have printed 10 lines on a page, and an SKP instruction was encountered, the assembler sends (length – 10 + 3) CR/LF pairs.

The SKP instruction is not required to cause pagination to occur when the standard length of a listing page is exceeded. Thus, if you are working with a default length of 60 for your standard length, then each 60 lines from the last page break forces a new page break.

Page Headings

The heading for each listing page is —

```
MODULE: {name}
```

where {name} is the name of the module currently being assembled. This heading can have additional information added to it through the HED pseudo-instruction. This instruction has the form —

```
HED {comment}
```

When this instruction is encountered, and a listing is being generated, pagination immediately occurs, the same as with the SKP instruction (see above). On the new page, and on all pages after it, the indicated {comment} appears after {name} in the heading, replacing any previous information specified by an earlier HED instruction.

You can change the heading any number of times in a listing. This is frequently done in order to generate documentation by sections, even though all sections may reside in a single module.

The heading appears on the page exactly the same as in {comment}, including the positioning of blanks, control characters, etc.

Blank Line Generation

If occasional blank lines are desired in a listing (usually to set off sections of code, or comments), they may be generated by including —

```
SPC {number}
```

at the desired spot in the source statements. {number} designates the number of blank lines desired. {number} can be any absolute expression, provided the expression evaluates to a positive integer (see “Symbolic Operations” below).

Non-Listable Pseudo-Instructions

The following pseudo-instructions do not appear in a listing —

```
LST  
UNL  
SKP  
HED  
SPC
```

Conditional Assembly

For reasons of complexity or length, it is occasionally desirable to selectively assemble only parts of a module. This is particularly true during the debugging stage of longer, complex assembly programs. “Conditional assembly” is the ability to designate certain portions of a module for assembly, depending upon conditions established by the IASSEMBLE statement.

You may recall from the description of the IASSEMBLE statement earlier, there are options called “conditions” available with the statement. These conditions —

```
A  
B  
C  
D  
E  
F  
G  
H
```

are used to designate which conditions are “set” during the assembly. By including one or more of these conditions, all conditional assembly statements predicated upon that condition are assembled. For example, if the following statement is executed —

```
IASSEMBLE Retrieve;A
```

then any occurrence of conditional assemblies based on “A” are assembled. Also, any conditional assemblies based on B through H are not assembled, since those conditions were not included in the options for the IASSEMBLE statement.

The conditional assembly sections are delimited by pseudo-instructions. A conditional section begins with one of the following —

```
IFA
IFB
IFC
IFD
IFE
IFG
IFH
```

and it concludes with —

```
XIF
```

In addition to the lettered conditions, a numeric condition can be tested by using an IFP pseudo-instruction. It has the form —

```
IFP {absolute expression}
```

The condition is considered true if {absolute expression} evaluates as a positive value. It should be noted that this is an assembly-time construct, meaning that the variables contained in the expression are evaluated at the time of assembly.

The IFP instruction performs in the same manner as the IFA through IFH instructions. It also terminates with the XIF instruction.

The conditional assembly is based upon a flag. At the beginning of the assembly for a module the flag is set so that object code is generated for all instructions. An IF conditional encountered during the assembly which does not have its condition set turns off the flag so that no further code is generated. Encountering an XIF statement resets the flag so that code generation can resume. For instance, if the source is —

```

430 ISOURCE LDA =Array_type
440 ISOURCE LDB =Array
450 ISOURCE JSM Get_info      ! Info on the array parameter
460 ISOURCE LDA Array_type    ! Look at the type
470 ISOURCE IFA               !
480 ISOURCE STA Test          ! DEBUGGING SECTION
490 ISOURCE RET 1             !
500 ISOURCE XIF               !
510 ISOURCE IFB
520 ISOURCE CPA =16           ! A file number (not an array)?
530 ISOURCE JMP ++3           ! Must be a file number
540 ISOURCE ADA =-12          ! Is it an array
550 ISOURCE SAP ++3           ! data type (ie, >12)?
560 ISOURCE XIF
570 ISOURCE LDA = Test

```

Then if —

```
IASSEMBLE Retrieve;A
```

is executed, lines 430 through 460, 480, and 490 are assembled, but 520 through 550 are not. Line 570 is assembled.

The one XIF actually affected both conditions. This effect is more dramatically illustrated by —

```
IASSEMBLE Retrieve
```

where neither A nor B is set. In this case 480, 490, 520 through 550 are not assembled. But 550 is assembled!

The effect of the XIF, then, is as a flag for all the conditions. As a consequence, it is not possible to “nest” conditional assemblies. This effect is the same with the IFP conditional.

Relocation

The code talked about in this section is relocatable. You do not have to worry about the absolute location of your module. The assembler automatically generates the appropriate machine codes for each of your instructions to assure that the correct location is reached when referenced.

Some instructions generate relocatable object code in which the operand address is an offset from the current address and the relocating loader has to make no changes to the object code for them as long as they are within -512 and $+511$ of the current address.

For indirect addressing, and for instructions which are more than 512 words away from the current address, it is required of the loader to adjust the address in the intermediate word to reflect the actual address being referenced. For indirect addressing generated by the assembler, this activity is automatic.

Some instructions permit you to specify an absolute machine address for its operand. In those cases, the assembler generates the code necessary to perform the reference to the absolute location.

For example, if the instruction was assembled —

```
LDA B
```

(which essentially says “load register A with the contents of register B) the result would be a machine instruction which references the B register (absolute address 1). This reference would be independent of the actual location of the instruction itself.

There are a couple of ways to produce an absolute address in an operand. The pre-defined symbols are one way. There is a type of expression known as “absolute” which is another way. Both of these are discussed in the next section, “Symbolic Operations”.

You should never try to use absolute addressing within the ICOM region, since not only is the location of the region itself not fixed, but modules can be moved around within the region.

Symbolic Operations

You have been introduced, in small doses, to symbols throughout the chapters preceding this one. The idea of symbols in an assembly language is the same as it is in a higher language such as BASIC — to make operations simpler and the code more understandable.

Several symbolic tools are provided for you in this assembly language system. You have already seen one described in detail in this chapter — labels. There are some pre-defined symbols the assembly system provides for certain locations in the machine (mostly registers). There are ways to define your own symbols (and give them a “type”). And, there are ways to access symbols in other modules.

Symbols can be used as operands in machine instructions and in some pseudo-instructions. They can be part of expressions in an operand.

Pre-Defined Symbols

The assembler has pre-defined a number of symbols and has reserved them as references to special locations in memory. Each of the locations has a special meaning and function. The symbols themselves are “reserved”, meaning they cannot be re-defined (by using them as labels on something else). The symbols are —

Symbol	Description
A	Arithmetic accumulator
Ar1	} BCD arithmetic accumulators
Ar2	
B	Arithmetic accumulator
Base_page	Global temporary area (50 words)
C	Stack pointer
Cb	Address-extension bit for byte pointer in C
D	Stack pointer
Db	Address-extension bit for byte pointer in D
Dmac	DMA count register
Dmama	DMA memory address register
Dmapa	DMA peripheral address register
End_isr_high	} Reserved symbols for writing interrupt service routines
End_isr_low	
Isr_flag	
Isr_psw	

Symbol	Description
Oper_1	} Arithmetic utility operand address registers
Oper_2	
P	Program counter
Pa	Peripheral address register
R	Return stack pointer
R4	} I/O registers
R5	
R6	
R7	
Result	Arithmetic utility result address register
Se	Shift-extend register
Ut1count	} Reserved symbols for writing utilities
Ut1end	
Ut1temps	

The meaning of each of these locations is discussed in other chapters. The absolute locations of the registers can be found in Chapter 2. A description of the function of the accumulators and pointers can be found in Chapter 3 as part of the discussion on machine instructions. A discussion of the I/O registers and symbols can be found in Chapter 7. The arithmetic registers are discussed in Chapter 5.

Using a pre-defined symbol in a machine instruction is the same as using its address. For example —

```
ISOURCE  LDA B
```

means simply that register A will be loaded with the contents of register B. The same effect could have been achieved with —

```
ISOURCE  LDA 1
```

except that the symbolic form makes it more obvious what is intended by the operation. This is true with most symbols.

Defining Your Own

You are defining your own symbol each time you specify a label on an instruction or pseudo-instruction. Normally the “value” of the label is the address associated with the instruction. However, in two cases it is possible to create the label and specify what its value is to be. One case is when the label is on the EQU pseudo-instruction; the other case is when the label is on the SET pseudo-instruction.

The EQU is an assembly-time construct. It exists only at the time of assembly to give you value-assigning capability to symbols. It generates no code itself, and it has no implementation or “location” in the object module.

To define a symbol using an EQU, the form is —

```
{label}: EQU {expression}
```

the resulting symbol ({label}) has the same “type” as the expression (see “Expressions” below) and it has the same value as the result of the expression.

As an example, assembling the statement —

```
ISOURCE Three: EQU 3
```

means that in all references in the module to the symbol “Three”, it is the same as referring to the **value 3**. Thus —

```
LDA Three
```

means load A with the contents of location 3.

A common use for this instruction is to assign a symbol an address which is an offset from another address. For example, if this sequence were in a module —

```
ISOURCE Save_registers: BSS 40B
ISOURCE Save_b: EQU Save_registers+1
```

then Save_b would refer to the second word in the BSS area “Save_registers”, and it would probably be used to store away the contents of the B register sometime —

```
ISOURCE STB Save_b
```

and later retrieve the value —

```
ISOURCE LDB Save_b
```

The SET pseudo-instruction defines a symbol in identical fashion to an EQU. Consequently, it has the same general form —

```
{label}: SET {expression}
```

The difference between the two is that the SET instruction can have its {label} be a symbol which has been previously defined. The effect in that case is to allow a **redefinition** of the symbol. For example, after assembling the following instructions —

```
ISOURCE Three: SET 3
ISOURCE Three: SET 30B
```

the symbol “Three” has the value 30B.

Literals

Literals are a special means of defining your own symbols without actually having to go to the trouble to do so. The result is a form of symbolic addressing without the symbol.

The form of a literal is —

```
= {expression} [ , {expression} [ , ... ] ]
```

where {expression} may be any absolute or relocatable expression (see “Expressions” below).

Evaluation of Literals

When a literal is encountered in an operand, three things occur —

1. The literal is converted to its binary value. If there is more than one expression in the literal, then they are all converted.
2. The binary value is stored in a literal pool. If there is more than one expression in the literal, then they are stored contiguously in the order specified.
3. The address of where the value is stored is then substituted for the literal in the operand.

If the same literal is used in more than one instruction, only one value is generated in the literal pool. All instructions using this literal refer to the same location.

Literals can be part of expressions as well as having expressions as part of them. Since they ultimately are replaced by an address (pointing to a specific location within a literal pool), their “type” is “relocatable”. See the section on “Expressions” later in this Chapter.

Basically, a literal means “the address of {expression}”. An example should help in the understanding of literals. Suppose that you want to store the value 1 into the A register. There are two ways you could accomplish that purpose. You **could** code —

```
One: DAT 1
      .
      .
      .
      LDA One
```

or, you could use a literal and code —

```
LDA =1
```

Using the literal method is easier and is more self-documenting. While the literal form strictly says “load A with the contents of the **address** of the constant 1”, it can also be read as “load A with the constant 1”, and this short-hand version can be an excellent way of self-documenting your programs, not to mention the elimination of a lot of unnecessary symbols.

Nesting Literals

Since literals use expressions, and literals may be used in expressions, it is possible to have a literal within a literal (nesting). In fact, it may be done to any depth, though the most useful form of nesting is a single level.

Suppose you want to initialize a variable to the value of pi each time you enter a routine. A nested literal would be a way of accomplishing this in a clean, straight-forward fashion —

```
Pi: BSS 4
    .
    .
    .
    LDA ==3.14159265349
    LDB =Pi
    XFR 4
```

and the locations starting at “Pi” now contains the full-precision value indicated (which is a fair approximation to pi). This would replace coding which could have looked like this (without using literals) —

```

A_init:  DAT Init
Init:    DAT 3.14159265349
A_pi:    DAT Pi
Pi:      BSS 4
        .
        .
        .
        LDA A_init
        LDB A_Pi
        XFR 4

```

Nonsensical Uses of Literals

A literal, basically, is an address. Since it can be used in an operand wherever an address may be used, it is possible to use it in instructions where the result is a little nonsensical.

For example, consider the result of doing some of the following —

```

STA =2
JSM ="GARBAGE"
DSZ == 1
JMP =Neverneverland
SZA =Out_to_lunch

```

Caution dictates that you well consider the appropriateness of the action when using the literal. Literals can be a highly useful tool, but only when properly employed.

Literal Pools

Literals are assemble-time constructs, but they eventually resolve to an actual address in the object code. That address points into a literal “pool”.

A literal pool is part of your module where the actual values of literals are stored. There is automatically a literal pool assigned at the end of each module where literals are used. As many literal values as possible are stored there by the assembler. However, in some cases, a literal pool is needed earlier in the program (a need indicated by the assembler with the “LT” assembly-time error). In that case a pool should be created using the LIT pseudo-instruction. This instruction has the form —

```

LIT {size}

```

where {size} is the number of words to be set aside (it may be a positive numeric expression). The instruction acts very much like a BSS. And, like a BSS, it should be placed at a location in your code where it is not likely to be inadvertently executed.

Most modules do not need assignment of an extra literal pool. However, one is needed where there is a literal used beyond 512 words from the first available space in the literal pool at the end of the module. To alleviate the problem, a literal pool must be created with the LIT statement within 512 words of the instruction.

A common cause of this kind of problem is a large BSS assignment between the instruction and the end of the module. Sometimes moving the BSS to some other location is a solution to the problem.

Expressions

Literals, some pseudo-instructions (particularly EQU), and a number of machine instructions, all permit “expressions” to be used as an operand. These expressions take one of two forms — “absolute” or “relocatable”. The type of an expression depends upon the type of the individual **elements** in it.

An element is of the type “absolute” if it is any of the following —

- A decimal integer (like 0, 1, 2, 1 024).
- An octal integer (like 10B, 40B, 100000B).
- A string (enclosed by quote marks) (like “ERROR”)
- An ASCII character, preceded by an apostrophe (like 'A).
- A label associated with an EQU or SET pseudo-instruction whose expression is also evaluative as type absolute (like EQU 40B).

An element is of the type “relocatable” if it is any of the following —

- A label not associated with an EQU or SET pseudo-instruction (i.e., it is an “address”).
- A literal (like =0).
- An asterisk, symbolizing “current address”.
- A label associated with an EQU or SET pseudo-instruction whose expression is also evaluative as type relocatable (like EQU *).

An expression is a list of elements each pair of which is separated by one of the following operators —

+ - / *

meaning addition, subtraction, division, and multiplication, respectively, as in BASIC.

The result of an expression is either absolute or relocatable depending upon the following rules:

An absolute expression is any expression which contains —

- Only absolute elements.
- An even number of relocatable elements, paired in sequence and by sign (i.e., for each relocatable element there is another relocatable element adjacent to it, of opposite sign). These pairs may be in combination with absolute elements.

A relocatable expression is any expression which contains —

- An odd number of relocatable elements, paired in sequence and by sign, except the last, which must be positive.
- An odd number of relocatable elements, as above, in combination with any number of absolute elements.

Any combination of absolute or relocatable elements which does not result in either an absolute or relocatable value, by the rules above, results in an error.

These rules and the rules for using * and / can be summarized as —

The expression is —	The type is —	Example
absolute ± absolute	absolute	1000B + 10
absolute + relocatable	relocatable	1 + Temp
relocatable ± absolute	relocatable	Temp - 1
relocatable - relocatable	absolute	Temp 1 - (Temp - 1)
relocatable + relocatable	error	Temp + Temp 1
absolute - relocatable	error	1000B - Temp
absolute * absolute	absolute	100 * 3
absolute / absolute	absolute	100 / 3
absolute * relocatable	error	Temp * 3
relocatable * absolute	error	3 * Temp
absolute / relocatable	error	Temp / 3
relocatable / absolute	error	3 / Temp

Unlike BASIC, there is no precedence among the operators. All are of equal precedence. Where precedence is desired, parentheses must be used. So where BASIC requires —

$$2*16+3*8$$

to result in 56, the same expression in the assembly language results in 280 (assembly language operators are evaluated from left to right). However, 56 would be the result if it were expressed as —

$$(2*16)+(3*8)$$

An expression may be of any length and contain as many operators and parentheses as desired, as long as the result can be evaluated and the parentheses are properly paired. All operators are evaluated from left to right. Multiplication and division can only be used with elements that are of type absolute.

External Symbols and Elements

There is an additional relocatable element, called “external”. It behaves in almost all respects as does any other relocatable element, except that only one external item may appear in an expression. Also, the expressions containing —

$$\text{relocatable} - \text{relocatable}$$

are not allowed when one of the relocatable elements is external. Externals are defined as symbols appearing in an EXT pseudo-instruction —

```
EXT {symbol} [, {symbol} [, ...] ]
```

These are entry points in another module or utility. “Entry points” are merely symbols in a module which are listed in an ENT pseudo-instruction in that module —

```
ENT {symbol} [, {symbol} [, ...] ]
```

If one module contains —

```
ENT Stage_left
```

then that symbol would be available to another module which contains —

```
EXT Stage_left
```

At execution time for a module with EXT instruction, all of the symbols listed in it must be either a utility name or be contained in an ENT or SUB (described in Chapter 6) of another module. It is not necessary that the module be in source form; it may already be an object module assembled from a source module which contained the symbol as an ENT or SUB.

Other Absolute Elements

There are additional **absolute** elements which may be used in expressions. These are “machine addresses”, short-precision numbers, and full-precision numbers.

A machine address is one of the following —

- An assembler pre-defined symbol.
- A symbol associated with an EQU or SET pseudo-instruction whose expression is evaluated as a machine address (i.e., it contains a pre-defined symbol or another EQU-associated symbol whose expression contains a pre-defined symbol).

For the most part, machine addresses can be used just like absolutes. However, they remain defined from assembly to assembly. By defining a machine address in one module (with an EQU or SET), it then becomes available to you with the same value in other modules which you assemble.

For example, if you were to assemble a module containing —

```
ISOURCE R100: EQU A+100
```

then R100 is a machine address following the above rules, just as if the assembler had pre-defined it. If you don't do any SCRATCH or GET statements in the meantime, then the next assembly you do would also have this symbol available without ever having to define it.

When full-precision numbers (like -2.5 , $3E3$, 3.141592) and short-precision numbers (like $1S$, $-2.5S$, $3.14159S$, $3E3S$) are used in expressions, they become the **entire** expression. This is because these numbers are only intended as simple data-generating devices in literals and in DAT pseudo-instructions. Explicitly, the rules for using full- and short-precision numbers are —

- They may only appear alone in an expression, i.e., they may not be in combination with other elements.
- They may only appear in literals and in DAT pseudo-instructions.

Utilities

A number of utilities have been provided to help make your programming tasks easier and to give you direct access to some of the operating system's capabilities and routines.

Descriptions of the utilities are made in conjunction with those topics where the utilities play a part. The form of the description of a utility is somewhat standardized. Each description will tell you —

- The name of the utility.
- The general procedure for using the utility.
- Any special requirements which must be satisfied for the utility to work properly.
- A step-by-step calling procedure for the utility.
- The exit conditions.

Utilities are a form of subroutine, so to execute them it is necessary to execute a jump-to-subroutine instruction (JSM) if you want the utility to return to the routine which calls it. Most utilities execute a RET 1 instruction to return, so in some cases where you follow a utility call with a RET 1 of your own, you can save the RET instruction by using the JMP (unconditional branch) instruction instead. For example, a typical utility call looks like —

```

"
"
"
LDA =Temp
LDB =Pointer
JSM Get_element
"
"
"

```

but if it happened to be followed by a RET 1 —

```

"
"
"
LDA =Temp
LDB =Pointer
JSM Get_element
RET 1
"
"
"

```

the calling procedure could be changed to —

```

*
*
*
LDA =Temp
LDB =Pointer
JMP Get_element
*
*
*
```

and you save a word of code: the effect is otherwise the same. Check the exit conditions for a utility before using this approach.

Utilities which you use in a module must have their names in an EXT pseudo-instruction for that module. Otherwise, the assembler is unable to tell that you meant a utility and not one of your own labels, causing an “undefined reference” assembly error.

Appendix F contains a short description of the utilities and has cross-references to the location in the manual of the full discussion on each utility.

The utilities currently available are —

Utility	Description
Busy	Tests the busy bits of a BASIC variable
Error_exit	Aborts an ICALL statement with a particular error number
Get_bytes	Accesses substrings (or parts of parameters)
Get_elem_bytes	Same as “Get_bytes”, but used for array elements
Get_element	Same as “Get_value”, but used for array elements
Get_file_info	Accesses the file-pointer of an assigned file
Get_info	Returns the characteristics of a variable passed as a parameter or existing in common
Get_value	Returns the value of a BASIC variable
Int_to_rel	Data type conversion from integer to full-precision
Isr_access	Establishes hardware linkages for interrupts
Mm_read_start	Prepares to read a physical record from mass storage
Mm_read_xfer	Reads a physical record from mass storage
Mm_write_start	Writes a physical record to mass storage
Mm_write_test	Verifies a physical record was written to mass storage
Printer_select	Changes or interrogates select-code for standard printer
Print_string	Outputs a string to the standard printer
Put_bytes	Replaces substrings (or parts of parameters)
Put_elem_bytes	Same as “Put_bytes”, used for elements in an array
Put_element	Same as “Put_value”, used for elements in an array
Put_file_info	Manipulates the file-pointer of a file
Put_value	Changes the value of a BASIC variable
Rel_math	Provides access to all the arithmetic routines
Rel_to_int	Data type conversion from full-precision to integer
Rel_to_sho	Data type conversion from full-precision to short
Sho_to_rel	Data type conversion from short-precision to full

Chapter 5

Table of Contents

Arithmetic

Binary Coded Decimal	83
Arithmetic Machine Instructions	84
BCD Registers	84
BCD Arithmetic	84
Addition	85
Ten's Complement for BCD	86
Floating Point Summations	88
Normalization	89
Rounding	89
Floating Point Multiplication	90
Floating Point Division	92
The FDV Instruction	94
Thirteen-Digit Dividends	95
Floating-Point Division Example	96
Arithmetic Utilities	99
Utility: Rel_math	99
Utility: Rel_to_int	102
Utility: Rel_to_sho	103
Utility: Int_to_rel	104
Utility: Sho_to_rel	105

Chapter 5

Arithmetic

Summary: Arithmetic operations are reviewed and the arithmetic utilities are discussed. Floating point and BCD arithmetic are explained.

Numerical calculations are a large part of any computer's operations. Implemented within the 9835A/B's processor are both integer and primitive Binary Coded Decimal (BCD) floating-point arithmetic operations. This chapter deals with those operations and is intended for those readers who may have no acquaintance with this topic, or perhaps only a passing one. The particular machine instructions involved with such arithmetic are reviewed.

Because the processor provides only rudimentary floating-point operations and because complete floating-point operations (e.g., subtract, divide) are not easy to write, utilities have been provided to perform these calculations. These utilities are discussed later in this chapter. If you are not interested in doing your own BCD arithmetic, it is recommended you skip immediately to "Arithmetic Utilities".

Binary Coded Decimal (BCD) uses four-bit binary codes to represent decimal digits. Thus, the 12-digit mantissa of a full-precision number is represented by 48 bits. The BCD digits are as follows —

DECIMAL	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

A BCD number within this manual has its digits represented as D_1 , D_2 , D_3 , etc., with each digit corresponding to some BCD digit. D_1 is the most significant digit in a number. Since full-precision numbers within the 9835A/B contain 12-digit BCD mantissas, 12-digit BCD numbers are used as the most frequent examples in this discussion. In that case, D_{12} is the least significant digit in a number.

Arithmetic Machine Instructions

There are some machine instructions which specifically operate upon the BCD registers. The discussions in this chapter will make use of the capabilities of these instructions to develop the techniques to write BCD arithmetic routines. If you have not done so already, you should familiarize yourself with the instructions before moving on in this chapter. A description of the instructions can be found in “Arithmetic Group” in Chapter 3.

BCD Registers

There are two registers in the machine used for BCD arithmetic — Ar1 and Ar2. These symbols are pre-defined by the assembly language to the registers’ locations in memory (see Chapter 3). The mnemonics for some instructions occasionally refer to these registers as X and Y respectively (see Chapter 3).

BCD Arithmetic

To understand BCD arithmetic in the context of the 9835A/B, recall from Chapter 3 that a full-precision value is represented in four words which contain its information as follows —

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit
Exp	Exponent										0	0	0	0	0	Man
Sign																Sign
D_1 (most significant digit)				D_2				D_3				D_4				
D_5				D_6				D_7				D_8				
D_9				D_{10}				D_{11}				D_{12} (least significant)				

Ten's Complement for BCD

The addition of the ten's complement of a number is used in lieu of a subtraction mechanism. If the signs of the two numbers to be summed are different, one of the numbers is complemented (it doesn't really matter which one), before the addition.

The ten's complement of a number with n digits to the left of the decimal point is —

$$X = 10_n - X$$

The ten's complement of a floating-point number has the same exponent as the original number. Since the mantissa (M) of a full-precision number can be assumed to have the decimal point implied after D_1 , then the number must be less than 10 (but greater than 0) and the ten's complement of a mantissa becomes —

$$M = 10 - M$$

Accordingly, all that is necessary to complement a floating-point number is to complement the mantissa. It is immaterial whether the mantissa is treated as a 12-digit integer or as a number between 0 and 10; the same sequence of digits results.

There are two instructions for doing ten's complements — CMX and CMY. The only difference between them is that CMX operates on the Ar1 register and CMY operates on the Ar2.

CMX and CMY leave the exponent word of a full-precision number completely alone. This means that the sign of the mantissa and the entire exponent are left unchanged in a ten's complement by CMX and CMY.

Ten's complement helps to accomplish addition, too. Rather than go into all of the nuances and subtleties of the arithmetic process, there is a simple rule for accomplishing decimal summations using ten's complements. Assuming the exponents are the same for the numbers to be added —

- If the signs of the numbers are the same, simply add them and leave the signs alone. If DC occurs, the result (Ar2) must be shifted to the right one place, and the exponent adjusted.
- If the signs of the numbers are different, complement, then add. A further complementing action may be necessary: if DC occurs, then the result necessarily has the same sign as the number which was not complemented; if DC does not occur, then the result must be complemented and then given the sign of the number which was complemented.

The FXA instruction is used to add mantissas. Here is a routine to implement the rule —

```

ISOURCE LDA Ar1          ! Check the sign
ISOURCE ADA Ar2
ISOURCE SLA Just_add      ! Skip if they are the same
ISOURCE CMX              ! Complement Ar1
ISOURCE FXA              ! Add the mantissas
ISOURCE LDB Ar2
ISOURCE SDS ++3          ! Was there an overflow?
ISOURCE CMY              ! No, so complement result
ISOURCE LDB Ar1          ! and switch exponents and signs
ISOURCE STB Ar2          ! Store the larger sign
ISOURCE JMP Done
ISOURCE Just_add: ! Do the addition
ISOURCE FXA
ISOURCE SDC Done         ! Was there an overflow?
ISOURCE LDA =1          ! Yes, so shift in a 1
ISOURCE LDB =1          ! into the most
ISOURCE MRY             ! significant digit
ISOURCE LDA Ar2         ! Adjust exponent
ISOURCE ADA =100B
ISOURCE STA Ar2
ISOURCE Done: ! CONTINUE ON

```


Floating Point Summations

In the example just completed, you may have noted that to copy the sign the entire exponent word was copied. What if the exponents were different? The answer is — the exponents must have been the same. In fact, the only reason the example worked at all was that the exponents were the same.

If exponents are different, addition of mantissas cannot proceed properly. To add the numbers it is necessary to make the exponents the same by shifting one of the mantissas an amount equal to the exponent difference.

This difference is easily found by subtracting the smaller exponent from the larger. If the difference is eleven or less (the precision of the 12-digit mantissa), it is possible to offset the mantissa of the number with the **smaller** exponent.

For example suppose there are two numbers to be added —

```
X.XXXXXXXXXXX E6
Y.YYYYYYYYYYYY E4
```

By shifting the smaller one to the right by 2 digits (the difference between 6 and 4), it is possible to align the exponents —

```
X.XXXXXXXXXXX      E6
0.0YYYYYYYYYYYYY E6
-----
Z.ZZZZZZZZZZZZ    E6
```

As can be readily seen from the example, a shift of more than 11 digits would cause the smaller value to be all zeroes in the significant 12 digits.

The digits to the right of the 12 most significant digits are lost in the action of shifting. That is, all except the left-most one. When using the MRX or MRY instructions, this digit is retained in the A register (bits 0-3) so that it can be used later for rounding purposes.

To use the MRX or MRY instructions, the number of digits to be shifted must be present in the B register.

The process for this “justification” of exponents can be summed up as follows:

- Subtract one exponent from the other storing the absolute value of the difference in the B register.
- Execute the MRX shift if the Ar1 register is smaller; execute the MRY shift if the Ar2 register is smaller.

Normalization

The raw result of an arithmetic operation (such as FXA) might not be a floating-point number that fits the standard form. It might have a leading DC needing to be incorporated into the number, as was seen in the “Addition” section earlier. Another possible deviation is a resulting D₁ of zero and no overflow. There could also be several zero-valued digits as left-most digits of the mantissa.

Such situations call for “normalization”. One type of normalization is accomplished with the NRM instruction. This instruction shifts register Ar2 left, leaving the number of shifts required in the B register as a binary number. The maximum number of shifts NRM performs is 12. If NRM must do all twelve shifts, Ar2 must have been 0. This is indicated by a value of 12 left in B and DC being set. For any other shift-count, NRM will leave DC at 0.

The rules for the normalization process are —

- Execute the NRM instruction.
- Follow this instruction by adding the complement of the contents of B (shifted left 6 bits) to the Ar2 exponent unless DC is set. If DC is set, store 0 into Ar2.
- Test the exponent result for an underflow.

Rounding

The addition operation (FXA) does not automatically round a result, and there is no instruction which does rounding in one step. Instead, it is necessary that a series of instructions be established to accomplish the result.

Recalling from “Floating Point Summations” (above) that the leftmost digit for rounding purposes (if any) is typically deposited in the A register by an MRX or MRY instruction, this digit can be checked to determine if rounding is required.

The process of rounding, then, would have the following steps —

- Determine from register A if rounding is required (i.e., if it's greater than or equal to 5).
- If rounding is not required, take no further action. If rounding is required, then load register B with 1 and execute an MWA instruction. This has the effect of incrementing the mantissa in Ar2 by 1. This action is an easier method than setting Ar1 to 1 and executing an FXA and it's faster, too. Don't forget to check DC for an overflow.
- One way the sequence of rounding could appear is —

```

10  ISOURCE  ADA =-5      ! Scale A down
20  ISOURCE  SAM +=3      ! If less than 5, no rounding
30  ISOURCE  LDB =1       ! Get ready to add 1 to Ar2
40  ISOURCE  MWA          ! Add 1 to least significant digit of Ar2

```

Floating Point Multiplication

Twelve-digit BCD floating-point multiplication is partially accomplished using the FMP instruction. This instruction effectively multiplies the value in the Ar1 register by a digit contained in B and adds the result to a partial product in Ar2.

Since, in the full multiplication process, exponents are merely added together, that part of the process is trivial. The ultimate sign of the product is also a trivial matter, determined by inspection of the signs of the original operands. Then the only matter of difficulty in the process is the actual multiplication of the mantissas. By way of explanation, assume that there are two mantissas to be multiplied —

multiplicand = A B C D

multiplier = W X Y Z

Just four digits are used to reduce the amount of symbolism required of the example. The same procedures and conclusions are applicable to a full twelve BCD digits.

One symbolic way to indicate how this multiplication is done is —

$$\begin{array}{r}
 \begin{array}{cccc}
 & A & B & C & D \\
 \times & W & X & Y & Z \\
 \hline
 & 0 & 0 & 0 & 0 & = \text{partial product 0} \\
 Z_{ov} & Z_1 & Z_2 & Z_3 & Z_4 & = Z(ABCD) \times 10^0 \\
 \hline
 & P_4 & P_5 & P_6 & P_7 & P_8 & = \text{partial product 1} \\
 Y_{ov} & Y_1 & Y_2 & Y_3 & Y_4 & 0 & = Y(ABCD) \times 10^1 \\
 \hline
 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8 & = \text{partial product 2} \\
 X_{ov} & X_1 & X_2 & X_3 & X_4 & 0 & 0 & = X(ABCD) \times 10^2 \\
 \hline
 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8 & = \text{partial product 3} \\
 W_{ov} & W_1 & W_2 & W_3 & W_4 & 0 & 0 & 0 & = W(ABCD) \times 10^3 \\
 \hline
 P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8 & = \text{partial product 4 (result)}
 \end{array}
 \end{array}$$

Notice that at each stage the multiple of ABCD, such as $X(ABCD)$, must be multiplied by an increasing power of ten in order that the digits of the multiple line up appropriately with the digits of the last partial product. An equivalent procedure is to have the partial product shifted right one digit at each stage.

Now, consider for a moment what is necessary within the assembly language to generate partial product $1 = 0 + Z(ABCD)$. Ar2 must be cleared and Ar1 is loaded with ABCD. Z is stored into B in bits 0 to 3. Then the FMP instruction is executed. Ar1 is added to Ar2 Z times, producing Z(ABCD) in Ar2. The overflow digit, Z_{ov} , ends up in the A register (bits 0 to 3). The overflow digit could be any value from 0 to 9 (each add could cause a carry, and there can be up to nine additions).

To create the next partial product, a mantissa right-shift on Ar2 must occur. Notice that mantissa right-shifting instructions (MRX and MRY) also shift bits 0 to 3 of the A register into D1. Thus, the right-shifting of the partial product (which must occur to prepare Ar2 for the next partial product) also automatically takes care of retaining the overflow digit.

Next, ABCD is added to $Z_{ov} Z_1 Z_2 Z_3$ a total of Y times (again by use of the FMP instruction). Partial product 2 is created. The process is repeated for the X and W digits, producing the result in Ar2.

After the final partial product has been calculated by the final execution of the FMP instruction, it is possible that a non-zero digit may be present in bits 0-3 of the A register. Such a digit is necessarily the most significant digit of the final product. In this case, another MRY execution is required. Further, the exponent of the product (which was initially estimated as the sum of the operand's exponents) must be incremented by one to reflect this power-of-ten shift.

Upon each step of partial product summation, a significant digit is lost due to the shift. This can't be helped. In general, the product of two 12-digit numbers has 24 digits of precision, but the bottom 12 digits must be discarded since only 12 BCD digits are stored in a mantissa. An error analysis of the algorithm discloses that dropping these digits causes the answer, on average, to be slightly smaller than it should be. However, rounding introduces a similar error, but in the other direction. Note that the process did not round each partial product.

The discarded digits can be inspected before they are permanently lost. The MRY instruction causes the digit to be placed in the A register (in bits 0 to 3). This provides an easy way for a rounding mechanism to check on those digits as they are discarded. The rounding routine needs to save the last digit discarded for use in rounding in the event the last use of FMP produces no overflow digit.

Finally, it should be noted that you can put WXYZ into B at the very start of the process and simply shift B right 4 bits (with an SBR 4 instruction) between each execution of FMP. After all, FMP uses only bits 0 to 3 of the register as the number of times to add Ar1 and Ar2.

Floating Point Division

There are many possible algorithms to accomplish floating-point division. The one presented here was chosen because of its effective use of the machine instructions and data structures employed by the processor and operating system.

Remembering that full-precision numbers consist of both a signed mantissa and a signed exponent, use can be made of the mathematical properties of both to reduce the division problem to manageable proportions. Suppose that you have two full-precision values to divide —

$$- 4.8E3 \div 1.5E - 2$$

The mathematical properties of exponents can be utilized and the second exponent can be subtracted from the first giving the exponent of the answer (subject to possible later adjustment). This is the first (and easiest) step in the division algorithm.

Secondly, the mathematical properties of signs within a division process can be used to determine the sign of the quotient from the signs of the divisor and dividend (negative quotient if the signs are different, positive quotient otherwise).

Thus, the problem can be reduced to the division of the mantissas —

$$(-4.8 \div 1.5) E5$$

As long as the full-precision numbers have been normalized, this adjustment of the exponents works for any pair of exponents. The normalization of the numbers also assures that the division of the mantissas under the following algorithm is sufficient to produce the mantissa of the result.

Since the decimal point of each mantissa is in the same place, they can be dropped altogether. For example —

$$-4.8 \div 1.5 = -48 \div 15$$

The algorithm can then consider both the divisor and the dividend as 12-digit integers.

The algorithm begins by placing the normalized values into the BCD arithmetic registers. The divisor (1.5E2 in the example) is transferred to register Ar1. The dividend (-4.8E3 in the example) is transferred to register Ar2. Basically, the algorithm subtracts the absolute value of the mantissa of Ar1 from the absolute value of the mantissa of Ar2 until Ar2 is smaller than Ar1. The number of subtractions required for that to occur becomes the first digit in the quotient (it'll be some value between 0 and 9 because the mantissas are normalized). If there is a (non-zero) remainder, then it is shifted left (multiplied by 10) and the subtraction process is repeated to calculate another digit in the quotient. The process is repeated until either a zero remainder occurs, or sufficient digits have been calculated, whichever occurs first. The resulting digits are merged, in order, to form the complete mantissa of the quotient.

There are some points to keep in mind in following the algorithm —

- Suppose you have a divisor whose normalized mantissa is larger than the normalized mantissa of the dividend, for example —

$$15 \div 48$$

then the first digit of the quotient's mantissa could easily be zero. If calculation of only twelve digits were made, the first digit being zero would mean a loss of a significant digit. To guarantee that there are always at least 12 significant digits calculated for the quotient, it is necessary (and sufficient) to calculate 13 digits. The 13th digit can always be thrown away, or used for rounding, if the first digit is not zero. Thirteen digits are always sufficient because you can never have a quotient with **two** leading zeroes, if the divisor and the dividend are both normalized.

- The number of subtractions during the calculation of any digit in the quotient is always nine or less. Again, this is true because the divisor is normalized and its first digit is always non-zero.
- At times during the algorithm, it is necessary to left-shift the mantissa of Ar2 (the mantissa at this point is the remainder). When shifting the remainder to the left (multiplying it by 10), you are shifting the first digit out of Ar2. If this digit is zero, this is not a problem. But, if the digit is non-zero, you can't ignore it during subtractions of the divisor. This in effect means that you are dealing with a 13-digit dividend! Since the machine instructions deal in 12-digit arithmetic, it is necessary that the algorithm handle the thirteenth.

The FDV Instruction

The FDV instruction provided by the processor is the primary tool used to implement the algorithm in assembly language. The instruction works by accomplishing the equivalent of automatically repeated subtractions of Ar1 (the divisor) from Ar2 (the dividend) until Ar2 is smaller than Ar1. The instruction actually adds the divisor to the ten's complement of the dividend until an overflow occurs. However, this is equivalent to subtracting until an “underflow” occurs. It is easier to understand the procedure if the discussion is in terms of “subtractions”, but it should be kept in mind that what is really occurring with the instruction is repeated “complement-additions” until overflow. This process is what is meant by the term “subtractions until overflow”.

The FDV instruction returns the number of subtractions without overflowing as a binary number in the B register (bits 0-3). The remaining bits in the B register (4-15) are cleared.¹ In effect, then B contains the next digit in the quotient.

This process is repeated for the number of digits to be calculated. After each FDV execution, the result of the overflow subtraction is left in Ar2. Since Ar2 does not contain the remainder, it is necessary to patch Ar2 so that it will contain the proper value for the next calculation. To get the proper value it is necessary to add Ar1 back into Ar2 to undo the results of the last subtraction (which caused the overflow).²

There is one case, however, where Ar2 does not need to be patched up, and this is when the remainder (Ar2) is zero. This situation implies not only that no patching up is needed, but also that the quotient is complete — no further digits need be calculated. It should be noted that the number of subtractions (which has been stored in the B register) is one count too small, thus B has to be incremented in this case so that it can be used as the last digit in the quotient.

Thirteen-Digit Dividends

The largest difficulty in the algorithm is attempting to deal with those instances where the dividend has thirteen digits. This situation arises when you shift the remainder left a place. The most significant digit must be retained when it is non-zero so that the subtractions are subtracted from the proper amount.

This shifting can be accomplished with the MLY instruction. With the way that the MLY instruction operates, the left-most digit (D₁) ends up being shifted out of Ar2 into register A (in the lower 4 bits, 0-3). Thus, the thirteen-digit algorithm must accommodate the most significant digit residing in the A register and the twelve least significant digits in the Ar2 register. The use of FDV must now take this modified situation into account.

When the FDV instruction is executed, Ar1 is subtracted from Ar2 until an overflow occurs. When this overflow occurs, it is necessary to decrement A and keep subtracting (without patching up Ar2). Each time an overflow occurs, A must be decremented until finally an overflow occurs when A is 0. This can be handled very neatly within a small loop.

¹ Since bits 4-15 of the register are cleared during execution of the FDV instruction, you can't accumulate quotient digits there. After each digit is calculated, it is necessary that you store the digit as part of a quotient which you keep stored in another location.

² This is equivalent to complementing Ar2, adding in Ar1, then complementing Ar2 again.

Another aspect of dealing with thirteen-digit dividends is the count placed in B with each execution of FDV. Since each overflow is a “successful” subtraction in the sense that is part of a proper count of subtractions (at least until A is 0), then that subtraction must be counted, too. The difficulty with this is that FDV does not count this last (overflowing) subtraction. The solution obviously is to add 1 to the value in the B register each time FDV causes an overflow. However, with the last overflow, being the “real” overflow, the 1 shouldn’t be added in, so after adding it in (during the loop), you have to subtract it back out again (after leaving the loop). To further complicate matters, if you have a zero remainder, you have to add it right back in again.

For example, if there happened to be three uses of FDV for a certain quotient digit, you form the quotient digit as —

$$Q_n = (B + 1) \quad + (B + 1) \quad + B$$

value after 1st use of FDV value after 2nd use of FDV value after final use of FDV

If the same general situation produced a zero remainder, then the quotient digit is formed as —

$$Q_n = (B + 1) \quad + (B + 1) \quad + (B + 1)$$

value after 1st use of FDV value after 2nd use of FDV value after final use of FDV

Floating-Point Division Example

An example of a 13-digit division routine follows. The rules which it implements are —

1. Always increment the value returned in B after an FDV operation.
2. After incrementing B, check the contents of A. If non-zero, loop immediately, performing no other tests or activities.
3. When a quotient digit has been found (i.e., A is zero), check to see if the remainder is 0. If so, exit the division loop. Save the last digit found as part of the answer.
4. If the remainder is not 0, decrement the value of the last quotient digit found and save it as part of the answer. Then add back the divisor to the remainder.

The example does not include routines for testing and handling —

- signs
- division by zero
- exponents
- overflow
- rounding

These have to be handled in a real program before or after the division algorithm itself (as appropriate).

```

ISOURCE ! Some useful symbols
      .
      .
      .
ISOURCE Ar21: EQU Ar2+1 ! First mantissa word
ISOURCE Ar22: EQU Ar2+2 ! Second mantissa word
ISOURCE Ar23: EQU Ar2+3 ! Third mantissa word
      .
      .
      .
ISOURCE ! Working area
ISOURCE Quotient: BSS 5 ! Working storage for quotient
ISOURCE Quotient_1: EQU Quotient+1 ! for quotient word 1
ISOURCE Quotient_2: EQU Quotient+2 ! for quotient word 2
ISOURCE Quotient_3: EQU Quotient+3 ! for quotient word 3
ISOURCE Quotient_4: EQU Quotient+4 ! for quotient word 4
ISOURCE Quotient_ptr: BSS 1 ! for quotient word 1
ISOURCE Digit_counter: BSS 1 ! total digits (1-13)
ISOURCE Within_word_ctr: BSS 1 ! digit counter (1-4)
      .
      .
      .
ISOURCE ! Dividend already in Ar2, divisor already in Ar1
ISOURCE Divide: ! START OF DIVISION LOOP
ISOURCE LDA =Quotient_1
ISOURCE STA Quotient_ptr
ISOURCE CLR 4 ! In case of early termination, zero
ISOURCE CMY ! Complement the dividend
ISOURCE LDA =13
ISOURCE STA Digit_counter ! Initializes digit count to 13
ISOURCE LDA =0 ! Initialize FDV repetition counter to 1
ISOURCE !
ISOURCE Next_word: ! WORKS ON NEXT SET OF 4 BCD DIGITS
ISOURCE LDB =4
ISOURCE STB Within_word_ctr ! Initialize intermediate counter
ISOURCE !
ISOURCE Next_digit: ! WORKS ON NEXT QUOTIENT DIGIT
ISOURCE SBL 4 ! Clear lower bits of B
ISOURCE STB Quotient_ptr,I ! Clear next storage word
ISOURCE !

```

```

ISOURCE !
ISOURCE Zero_remainder:      ! ZERO REMAINDER BEFORE 13th DIGIT?
ISOURCE     DSZ Digit_counter
ISOURCE     JMP Shift
ISOURCE     JMP Done
ISOURCE !
ISOURCE Shift_left:      SBL 4
ISOURCE Shift:      DSZ Within_word_ctr      ! Shift digits as necessary
ISOURCE     JMP Shift_left
ISOURCE Fdv_loop:      ! QUOTIENT CALCULATION
ISOURCE     FDV      ! Ar2=Ar2+Ar1 until overflow
ISOURCE     ADB Quotient_ptr,I      ! Merge new digit with rest of answer
ISOURCE     ADB =1      ! Increment the new digit
ISOURCE     STB Quotient_ptr,I      ! Save this state of the answer
ISOURCE     RIA Fdv_loop      ! Decrement and loop if non-zero
ISOURCE !
ISOURCE ! Check for a zero remainder
ISOURCE !
ISOURCE     LDA Ar21
ISOURCE     IOR Ar22
ISOURCE     IOR Ar23
ISOURCE     SZR Zero_remainder
ISOURCE !
ISOURCE ! No zero remainder, so divide again. But first restore dividend,
ISOURCE ! shift it left, and then find new FDV repetition count.
ISOURCE !
ISOURCE     CMY      ! Decomplement remainder (Ar2)
ISOURCE     FXA      ! Add back in divisor (Ar1)
ISOURCE     ADB =-1      ! Undo the increment
ISOURCE     STB Quotient_ptr,I      ! Save the corrected partial answer
ISOURCE     CMY      ! Complement the dividend
ISOURCE     LDA =0      ! Clear A
ISOURCE     MLY      ! Shift dividend left
ISOURCE     ADA =-9      ! Determine next repetition count
ISOURCE !
ISOURCE ! Bottom of loop maintenance follows
ISOURCE !
ISOURCE     DSZ Digit_counter      ! Decrement number of digits
ISOURCE     JMP Within_word
ISOURCE     JMP Done
ISOURCE !
ISOURCE Within_word:      ! DECREMENT POSITION WITHIN WORD
ISOURCE     DSZ Within_word_ctr
ISOURCE     JMP Next_digit
ISOURCE     ISZ Quotient_ptr
ISOURCE     JMP Next_word
ISOURCE !
ISOURCE Done:      ! STORE AWAY THE RESULT
ISOURCE     STB Quotient_ptr,I      ! Store last digits of quotient
ISOURCE     LDA =Quotient
ISOURCE     LDB =Ar2
ISOURCE     XFR 4      ! Transfer quotient from working storage
ISOURCE     ! to Ar2
ISOURCE     ! to Ar2
ISOURCE     NRM
ISOURCE     SZB Continue      ! Go on, if all is OK
ISOURCE !
ISOURCE ! If leading digit of quotient was a zero, then old digit 13 must
ISOURCE ! be saved as new digit 12
ISOURCE ! be saved as new digit 12
ISOURCE !

```

```

ISOURCE    LDA Quotient_4      ! Get digit 13
ISOURCE    AND #17B           ! Lower 4 bits only (in case Quotient_4
                                ! is used elsewhere for other thi
                                ! is used elsewhere for other things
ISOURCE    ADA Ar23            ! Add in new digit (old digit 12 was 0)
ISOURCE    STA AR23           ! Save the corrected quotient
ISOURCE    ! Proceed to adjust exponent accordingly
            .
            .
            .
ISOURCE    Continue:          ! Compute sign, etc.

```

Arithmetic Utilities

Now that you have been introduced to the complexities of BCD arithmetic and floating-point operations, this is the time to present an easier way of accomplishing these operations — the arithmetic utilities.

In order to make BASIC a useful programming tool, the operating system already contains a number of floating-point routines. Recognizing that BCD and floating-point arithmetic can be a difficult and laborious task to implement, the assembly language provides a utility by which the operating system mathematical routines can be accessed. There are also utilities for the conversion of numerical data types.

UTILITY: Rel_math

The Rel_math utility provides access to all of the system floating point routines and functions.

General Procedure: The utility is told the execution address of the desired routine or function and is also told the number of parameters. The parameters are floating-point values stored in full-precision form (4 words each). The result is a full-precision value.

Special Requirements:

- If one operand is passed to the utility, the **address** of the operand is stored in register Oper_1.
- If two operands are passed to the utility, the **address** of the **first** operand is stored in register Oper_1 (as above), and the **address** of the **second** operand is stored in register Oper_2.
- The **address** of where the result should be stored must be stored in the register Result.
- All operands and the result are full-precision values and require 4 words each.

- Values passed must make sense for the routine or function being called (e.g., Oper_2 should not point to a value of 0 when calling the division routine), or else an error results.
- The storage areas for the operands and the result must reside either in the ICOM region or in the Base_page register. Specifically, they cannot be specified as Ar1 or Ar2.

Calling Procedure:

1. Assure that Oper_1, Oper_2, and Result contain the proper addresses as above.
2. Load register A with the number of parameters required for the routine or function (see the table on next page). Note that some routines require this number to be complemented.
3. Load register B with the execution address of the routine or function (see the table on the next page).
4. Call the utility.

Exit Conditions:

- The result is placed into the 4 words starting at the address pointed to by the Result register.
- Register A contains 0 if no error is encountered during execution of the utility.
- Register A contains the error number should an error be encountered during execution of the utility.

Table 1. Routines, Addresses, and Parameters for Rel_Math Utility

Routine	Execution Address (LDB =)	Operands (LDA =)
Addition	30620B	2
Subtraction	30612B	2
Multiplication	30732B	2
Division	31100B	2
Exponentiation	34066B	2
DIV	32574B	2
MOD	32725B	2
SQR	31240B	1
INT	32637B	1
FRACT	33052B	1
EXP	33763B	1
LOG	33773B	1
LGT	34053B	1
PROUND	32015B	-2
DROUND	32037B	-2
ABS	32622B	1
SGN	33441B	1
PI	36057B	0
RND	33377B	0
RES	36077B	0
TYP	6733B	1
SIN	34003B	1
COS	34014B	1
TAN	33741B	1
ASN	34025B	1
ACS	34040B	1
ATN	33751B	1
ERRL ¹	61765B	0
ERRN ¹	61753B	0
DECIMAL ¹	162026B	1
IADR	162167B	-2
IMEM	162150B	-2
OCTAL	162105B	1
AND	31632B	2
OR	31647B	2
EXOR	31615B	2
NOT	31661B	1
Less Than (<)	31667B	2
Less Than or Equal To (<=)	31675B	2
Not Equal (<>)	31727B	2
Equal (=)	31717B	2
Greater Than or Equal To (>=)	31711B	2
Greater Than (>)	31703B	2

¹ These functions return an integer value which is stored in the second word of the four words reserved by Result.

By way of example, suppose you have established two full-precision values which need to be multiplied. The call to the Rel_math utility to accomplish the multiplication would look similar to this —

```

SOURCE ! Working storage
      *
      *
SOURCE Operand_1: BSS 4
SOURCE Operand_2: BSS 4
SOURCE Product:   BSS 4
      *
      *
SOURCE Multiply: ! MULTIPLY THE OPERANDS
SOURCE      LDA =Operand_1
SOURCE      STA Oper_1
SOURCE      LDA =Operand_2
SOURCE      STA Oper_2
SOURCE      LDA =Product
SOURCE      STA Result
SOURCE      LDA =2           ! Call the multiply routine
SOURCE      LDB =30732B
SOURCE      JSM Rel_math
SOURCE      SZA **2          ! Test for any errors
SOURCE      JSM Error_exit   ! Error encountered, so leave
      *
      *

```

Note in the last line of the example the call to the Error_exit utility (page 191) is made when register A is not zero. When this occurs, A contains the error number of the error encountered — ready-made for calling the Error_exit utility.

UTILITY: Rel_to_int

The Rel_to_int utility provides for the conversion of a full-precision value into an integer.

General Procedure: The utility is given the address of the location of the full-precision value and the address of the location where the integer is to be stored.

Special Requirements: The full-precision value must be within the range of integers (– 32 768 to + 32 767).

Calling Procedure:

1. Store the address of the full-precision value into register Oper_1.
2. Store the address of where the integer is to be stored into register Result.
3. Call the utility.

Exit Conditions: The overflow bit in the processor is set if the integer is outside the range of integers.

An example —

```

ISOURCE ! Working storage
ISOURCE Operand:  BSS 4    ! Contains a full-precision value
ISOURCE Value:    BSS 1    ! Contains an integer value
ISOURCE
ISOURCE
ISOURCE
ISOURCE LDA =Operand
ISOURCE STA Oper_1
ISOURCE LDA =Value
ISOURCE STA Result
ISOURCE JSM Rel_to_int ! Convert real to integer
ISOURCE SOC #+3        ! Check for overflow
ISOURCE LDA =20        ! Set error number to 20
ISOURCE JSM Error_exit ! and take error exit
ISOURCE
ISOURCE
ISOURCE

```

UTILITY: Rel_to_sho

The Rel_to_sho utility provides for the conversion of a full-precision value into a short-precision one.

General Procedure: The utility is given the address of the location of the full-precision value and the address of the location where the short-precision value is to be stored.

Special Requirements: A short-precision value requires 2 words to be stored.

Calling Procedure:

1. Store the address of the full-precision value into register Oper_1.
2. Store the address of the storage area for the short-precision value into register Result.
3. Call the utility.

Exit Conditions: No special exit conditions.

As an example —

```

ISOURCE ! Working storage
ISOURCE Operand: BSS 4      ! Contains full-precision value
ISOURCE Value:   BSS 2      ! Contains short-precision value
      .
      .
      .
ISOURCE      LDA =Operand
ISOURCE      STA Oper_1
ISOURCE      LDA =Value
ISOURCE      STA Result
ISOURCE      JSM Rel_to_sho ! Convert full to short
      .
      .
      .

```

UTILITY: Int_to_rel

The Int_to_rel utility provides for the conversion of an integer into a full-precision value.

General Procedure: The utility is given the address of the location of the integer and the address where the full-precision value is to be stored.

Special Requirements: None.

Calling Procedure:

1. Store the address of the integer into register Oper_1.
2. Store the address of the storage area for the full-precision value into register Result.
3. Call the utility.

Exit Conditions: No special exit conditions.

An example —

```

ISOURCE ! Working storage
ISOURCE Operand: BSS 1      ! Contains an integer
ISOURCE Value:   BSS 4      ! Contains full-precision value
      .
      .
      .
ISOURCE      LDA =Operand
ISOURCE      STA Oper_1
ISOURCE      LDA =Value
ISOURCE      STA Result
ISOURCE      JSM Int_to_rel ! Convert integer to real
      .
      .

```

UTILITY: Sho_to_rel

The Sho_to_rel utility provides for the conversion of a short-precision value into a full-precision one.

General Procedure: The utility is given the address of the location of the short-precision value and the address of where the full-precision value is to be stored.

Special Requirements: None.

Calling Procedure:

1. Store the address of the short-precision value into register Oper_1.
2. Store the address of the storage area for the full-precision value into register Result.
3. Call the utility.

Exit Conditions: No special exit conditions.

An example —

```

ISOURCE ! Working storage
ISOURCE Operand: BSS 2 ! Contains short-precision value
ISOURCE Value: BSS 4 ! Contains full-precision value
ISOURCE
ISOURCE
ISOURCE
ISOURCE LDA =Operand
ISOURCE STA Oper_1
ISOURCE LDA =Value
ISOURCE STA Result
ISOURCE JSM Sho_to_rel ! Convert short to real
ISOURCE
ISOURCE
ISOURCE

```


Chapter 6

Table of Contents

Communication Between BASIC and Assembly Language

The ICALL Statement	107
Corresponding Assembly Language Statements	108
Arguments	109
“Blind” Parameters	112
Getting Information on Arguments	113
Utility: Get_info	114
Retrieving the Value of an Argument	116
Utility: Get_value	117
Utility: Get_element	118
Utility: Get_bytes	119
Utility: Get_elem_bytes	120
Changing the Value of an Argument	122
Utility: Put_value	122
Utility: Put_element	123
Utility: Put_bytes	124
Utility: Put_elem_bytes	125
Using Common	127
Busy bits	130
Utility: Busy	131

Chapter 6

Communication Between BASIC and Assembly Language

Summary: This chapter discusses the techniques used to pass information to and from assembly language programs. Calling assembly language routines and passing parameters are presented, along with issues involved in using common. Applicable utilities are also discussed.

Once assembly language programs have been written, they are executed using the ICALL statement. This statement is very similar to BASIC's CALL statement for subroutines. In fact, the function it performs is nearly identical in effect — the only difference is that the target subroutine has been written in assembly language instead of in BASIC. The ICALL statement also provides a means to pass data between BASIC and assembly programs through its argument list. Data can also be passed through common.

The ICALL Statement

There are two ways to execute an assembly language routine. One way is as an interrupt service routine when an interrupt occurs on the select code to which the service routine has been linked. This way is discussed in Chapter 7. The other way is through executing an ICALL statement, either in a BASIC program or from the keyboard.

The syntax of the statement is —

```
ICALL {routine name} [ ( {argument} [, {argument} [, ...]] ) ]
```

{routine name} is the name of the assembly language routine to be executed. {argument} is a data item which has the same characteristics as an argument in BASIC's CALL statement — there may be constants, variables, or expressions. (How these items correspond to instructions in the assembly language will be discussed shortly.)

By way of example, suppose that you have an ICALL which is being used to call a sort routine and the routine was written in such a way as to require two arguments be passed to it — an array to be sorted and the number of elements to be sorted (in that order). Then the following would be valid calls to that routine —

```
ICALL Sort(Test(*),100)
ICALL Sort(Test$(*),Number)
ICALL Sort(Value(*),Events/DIV/2)
```

Upon executing the ICALL statement, execution in a program transfers to the routine named. Upon return from the routine, control is passed to the BASIC statement which follows the ICALL. This is identical in effect to the CALL statement in BASIC.

In executing the statement from the keyboard, the routine named is executed just as if it were used in a program. Upon return from the routine, control is passed back to the keyboard. This is unlike BASIC's CALL statement, which cannot be executed from the keyboard.

To execute a routine, whether it be from a program or from the keyboard, its object code must currently reside in the ICOM region.

Corresponding Assembly Language Statements

When the ICALL is executed, it references a routine in the object code. When the module containing the routine was assembled, it declared that routine name as a "subroutine" entry point. ("Subroutine" and "routine" are synonymous in this context.) This is done with a SUB pseudo-instruction and a label.

When a SUB pseudo-instruction appears in the source code, it is a signal to the assembler that a subroutine entry point follows. Then the first machine instruction (or some code-generating pseudo-instruction, such as BSS or DAT) must have a label. That label becomes the routine name. If the label is missing, an error results (assembly-time "SQ" error).

For example, in the above examples of ICALL, the Sort routine could have been defined by the sequence —

```
ISOURCE SUB
ISOURCE Sort: LDA #Array_info
```

except that there are arguments involved. (That exception is discussed in a moment.) The joint use of these two statements results in the label "Sort" being identified as a routine name, referenceable with an ICALL statement.

In general, no machine instructions or code-generating pseudo-instructions can be inserted between a SUB pseudo-instruction and the instruction containing the routine name. An exception to this exists when arguments are involved in a call.

Arguments

When a value is placed into an ICALL statement to be sent down to an assembly language routine, that value is called an “argument” (like the argument of a mathematical function). The corresponding structure on the assembly language side is called a “parameter”. A parameter “declaration” is an assembly pseudo-instruction by which a parameter is created.

When a routine is to be called with arguments, a parameter declaration pseudo-instruction is required for each one of the arguments. These declarations appear between the SUB pseudo-instruction and the instruction containing the routine name.

Thus, when there is a call like —

```
ICALL Sort(Test$(*),100)
```

the corresponding assembly language entry looks like —

```
ISOURCE      SUB
ISOURCE      STR (*)
ISOURCE      REL
ISOURCE Sort: LDA #Array_info
```

To accommodate the two arguments, two parameter declarations had to appear between the SUB instruction and the entry point. (In this example, they were the STR and REL declarations.) These declarations may even have labels of their own —

```
ISOURCE      SUB
ISOURCE Parameter_1: STR (*)
ISOURCE Parameter_2: REL
ISOURCE Sort: LDA #Array_info
```

The appearance of these labels does not effect the fact that “Sort” is the name of the routine.

Parameter declarations have “types” just like variables. These types have to correspond to the “types” of the arguments used in the ICALL. The declarations and their types are —

INT	meaning integer
REL	meaning full-precision
SHO	meaning short-precision
STR	meaning string
FIL	meaning a file number

In the above example, STR had to be used as the first parameter declaration because the first argument was a string. Similarly, REL had to be the second declaration because the second argument was a numeric expression (which is always full-precision).

When an array is to be passed, the declaration is followed by an “array identifier” — (*). Thus, when arrays are involved, the declarations appear as —

INT(*)	meaning an integer array
REL(*)	meaning a full-precision array
SHO(*)	meaning a short-precision array
STR(*)	meaning a string array

(File numbers do not come in arrays, so that declaration — FIL — cannot be followed by an array identifier.)

Since the example call above uses a string array as the first argument, the corresponding assembly language parameter declaration uses an array identifier after STR.

The parameter declarations are associated with the arguments in the ICALL in the same order. If the types do not match when the ICALL is executed, an error occurs (number 8).

So, if the subroutine entry looks like —

```

SOURCE      SUB
SOURCE      STR (*)
SOURCE      REL
SOURCE Sort: LDA =Array_Info

```


then this ICALL executes properly —

```
ICALL Sort<Test$(*),100>
```

but these ICALLs result in run-time errors —

```
ICALL Sort<Test$,100>
ICALL Sort<Test(*),100>
ICALL Sort<Test$(*),"ASCENDING">
```

Each declaration reserves three words in the object code upon assembly. As a result of the ICALL execution, these words contain a descriptor of the corresponding argument. These descriptors are used by the utilities for fetching and storing values. Thus, in the Sort calling example above, when the ICALL is executed, a descriptor for Test\$(*) is stored in the three words starting at Parameter_1. Similarly, a descriptor for the constant 100 is stored in the three words starting at Parameter_2.

The types discussed here do not apply just to simple variables, arrays, and constants. They also apply to single elements of arrays and expressions. If you have a STR parameter declaration, for example, any of the following would be valid as arguments in the ICALL statement —

```
Test$(1)
CHR$(127)&Test$
RPT$("A",20)
Test$[1,Stop]
```

It is similar for numerical expressions.

The number of arguments passed by an ICALL statement must be no more than the number of parameter declarations in the subroutine entry. There may be fewer, however. The actual number passed is stored in the word reserved by the SUB pseudo-instruction.

Unlike the CALL statement in BASIC, the ICALL statement can be executed from the keyboard. In doing so, any variables used as arguments pass their current values to the routine, rather than resetting them to 0 (this is the same contrast as between running a program by pressing **RUN** and running it pressing **CONT**).

“Blind” Parameters

With explicit parameter declarations, an error occurs if a different type of variable or expression is passed. In many cases, the error is desirable — you do not want different types of arguments corresponding to a single parameter declaration. But in other cases, the error might not be as desirable. Take the example of a sort. You might want the sort to have the capability of sorting any type of array. You have two choices in that case — you can make different routines, each with the appropriate declarations, or you can use a single entry point and the ANY parameter declaration.

The ANY declaration —

```
ANY
```

is “blind” to the type of the corresponding argument in the ICALL statement. When used, it accepts any type of argument as valid — string, full-precision, short-precision, integer, file number, array. The descriptor for the argument is stored in the three words set aside, just as in the other declarations.

Now, if your entry looks like —

```
ISOURCE      SUB
ISOURCE      ANY
ISOURCE      REL
ISOURCE Sort: LDA #Array_info
```

then any of the following calls would be valid —

```
ICALL Sort(Test$(*),100)
ICALL Sort(Test(*),100)
ICALL Sort(Test$,100)
ICALL Sort(Test,100)
ICALL Sort(#1,100)
```

When using the ANY declaration, it becomes the responsibility of your assembly language routine to determine what is a valid parameter and what is not. You lose the automatic type-checking available with explicit declarations. Techniques for doing this are discussed in the next section.

Getting Information on Arguments

When an ICALL is executed with an argument, and the corresponding parameter is blind, then it may be necessary for the purposes of your routine to know what type of argument is actually passed. This need can be present even when one of the explicit type declarations is used, since an expression or constant can be passed as easily as a variable.

A utility has been provided for obtaining this information, along with other “vital statistics” which may be useful to know during the execution of your routine. Before describing the utility itself, let’s look at the information which it can provide you about an argument.

The information returned by the utility is stored in an area which you set aside for it. The size of the area can vary from 3 words to 30. The information, when returned, is in the following form —

Word #	Description
0	Argument type (see description later)
1	Number of dimensions (0 for non-arrays)
2	Size, in number of bytes (dimensioned length, for strings)
(for arrays only:)	
3	Total number of elements in array ¹
4	Lower bound of first dimension ¹
5	Absolute size of first dimension (upper bound – lower + 1)
6	Lower bound of second dimension (if any) ¹
7	Absolute size of second dimension
8	Lower bound of third dimension (if any) ¹
9	Absolute size of third dimension
10	Lower bound of fourth dimension (if any) ¹
11	Absolute size of fourth dimension
12	Lower bound of fifth dimension (if any) ¹
13	Absolute size of fifth dimension
14	Lower bound of sixth dimension (if any) ¹
15	Absolute size of sixth dimension
16	Element offset
17	Size, in words, of each element (dimensioned length, for strings)
(dependent upon memory size of your machine:)	
18-20	Pointer parameters
21-23	Pointer parameters (only for machines over 64K bytes)
24-26	Pointer parameters (only for machines over 128K bytes)
27-29	Pointer parameters (only for machines over 192K bytes)

¹ Stored as a negative number.

The argument type returned in word 0 is as follows —

Value	Type
0	String expression
1	Full-precision expression
2	Short-precision expression
3	Integer expression
4	String simple variable
5	Full-precision simple variable
6	Short-precision simple variable
7	Integer simple variable
8	String array element
9	Full-precision array element
10	Short-precision array element
11	Integer array element
12	String array
13	Full-precision array
14	Short-precision array
15	Integer array
16	File number

The size, in bytes, will be one of the following values —

For an integer	2
Short-precision	4
Full-precision	8
String variables	dimensioned length
String expressions	actual length

The utility which retrieves all this information is called “Get_info”.

UTILITY: Get_info

General Procedure: The utility is told the location where the information is to be returned and the address of the parameter declaration. It returns with the information on the argument in the ICALL corresponding to the parameter declaration.

Special Requirements:

- The location where it is to store the information must be adequate to hold all that may be returned. For non-arrays, 3 words will suffice. For arrays, up to 30 words may be required (as above). If you are writing a general routine, it may be wise to play it safe by setting aside a full 30 words.
- An argument must have been passed by the ICALL (in the case of parameters) or a corresponding BASIC COM declaration must exist (in the case of common declarations).

Calling Procedure:

1. Load register A with the address of the storage area for the information to be returned.
2. Load register B with the address of the parameter declaration corresponding to the desired argument.
3. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the JSM. Since there are no error exits, and there is no requirement that there be as many arguments as there are parameter declarations, an argument must actually have been passed by the ICALL in order for the utility to work correctly.

Following up on the example in the previous section, suppose the first thing that the Sort routine does is check to see if the first parameter passed is an array. Then, by using the Get_info utility, it is possible to have the instructions look as follows —

```

ISOURCE Array_info: BSS 30
ISOURCE          SUB
ISOURCE Array: ANY
ISOURCE Number: REL
ISOURCE Sort: LDA #Array_info ! Get info on argument
ISOURCE          LDB #Array
ISOURCE          JSM Get_info
ISOURCE          LDA Array_info ! Get the argument's type
ISOURCE          CPA #16        ! Is it a file number?
ISOURCE          JMP Error_8    ! Yes, indicate error 8
ISOURCE          ADA #-12
ISOURCE          SAP #2         ! An array (types 12-15)?
ISOURCE          JMP Error_8    ! No, indicate error 8

```

The array information returned by the `Get_info` utility is used for accessing elements in arrays passed as arguments. It is used by the element-retrieval utilities described in a later section of this chapter. Once retrieved, the information is usable any number of times for accessing the array associated with it. It is not necessary to retrieve the information every time you access an array, as long as you have not altered the information (except the pointer) between accesses.

The seventeenth word of the array information (word 16 on the chart) is reserved to hold the offset from the start of the array of the element to be accessed. Therefore, it is permissible (indeed, it is **necessary**) to alter the contents of that location to indicate which element in the array you wish to retrieve. None of the other words returned by the utility should be changed.

In making multiple accesses with the same information, caution should be taken when an array is involved. If a `REDIM` statement is executed upon the array between accesses, the information may not reflect the true structure of the array. This potentiality can be addressed in one of two ways —

- Advise the BASIC user against using a `REDIM` on the array between executions of the routine or routines involved.
- Call the `Get_info` utility each time the array is accessed.

Similar problems exist when a BASIC subprogram is called recursively, and the subprogram uses a local array as an argument in an `ICALL`, or when a subprogram calls a routine and later exits (causing its local arrays to disappear).

Retrieving the Value of an Argument

At some point during execution of your assembly language routine, you may want to retrieve the value of an argument so that you can use it in your processing. By doing so, you accomplish one of the methods of communicating with assembly language — namely, passing a value TO the assembly language routine from BASIC.

There are a number of utilities for this purpose. The one to use is dependent upon the type of argument passed. The utilities available are —

Name	Used For
Get_value	Simple variables, expressions, individual elements of arrays passed as arguments, and file numbers
Get_element	Elements (from arrays passed as arguments)
Get_bytes	Substrings of strings passed as arguments either as simple string variables, expressions, or individual elements of arrays passed as arguments
Get_elem_bytes	Substrings of individual elements (from string arrays passed as arguments)

How each of these utilities is used is described in the immediately following pages.

UTILITY: `Get_value`

General Procedure: The utility is given the address of the parameter declaration and the address of where the value of the argument is to be stored. It returns with that value stored in the indicated area. It works on simple variables, expressions, and individual elements of arrays (passed as arguments), of any type.

Special Requirements:

- The storage area set aside for the value must be large enough to hold the value. The size of the storage area must be —

for a file number	1 word
for an integer value	1 word
for a short-precision value	2 words
for a full-precision value	4 words
for a string	maximum length in bytes $\div 2 + 1$ word (+ 1 additional word if the string length is odd)

- An argument must have been passed by the ICALL for the utility to work properly.

Calling Procedure:

1. Load register A with the address of the storage area for the value.
2. Load register B with the address of the parameter declaration.
3. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call.

Here is an example call to the utility, retrieving information from a full-precision argument —

```

ISOURCE Value:      BSS 4
ISOURCE              SUB
ISOURCE Parameter:  REL
ISOURCE Entry:      LDA =Value
ISOURCE              LDB =Parameter
ISOURCE              JSM Get_value
  
```

UTILITY: Get_element

General Procedure: This is similar to the “Get_value” utility. This utility retrieves a value from an element of an array passed as an argument. It works on arrays of any type.

Special Requirements:

- The storage area set aside for the value must be large enough to hold the value. Resultant, the size of the storage area must be —

for an integer	1 word
for a short-precision value	2 words
for a full-precision value	4 words
for a string	maximum length in bytes 2 + word (+ 1 additional word if the string length is odd)

- The array information must be retrieved with the “Get_info” utility before calling this utility.
- The offset of the element in the array must be correct in the array information (word 16 returned by “Get_info”). It should be remembered that the offset of the element is dependent upon the number of dimensions in the array and the length of each. A calculation may be necessary to arrive at the offset when accessing multiple-dimension arrays. The offset is in terms of number of elements.

Calling Procedure:

1. Store the element offset within the array information (word 16 returned by “Get-info”).
2. Load register A with the address of the storage area for the value.
3. Load register B with the address of word 0 of the information returned by the “Get_info” utility (see description of that utility).
4. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call.

Here is an example call, retrieving the third element (relative element 2) of an integer array and placing it into Value —

```

ISOURCE Value:      BSS 1
ISOURCE Array_info: BSS 30
ISOURCE Element:    EQU Array_info+16 ! Element offset
ISOURCE             SUB
ISOURCE Parameter:   INT (*)
ISOURCE Entry:       LDA =Array_info    ! Get the array info
ISOURCE             LDB =Parameter
ISOURCE             JSM Get_info
ISOURCE             LDA =2              ! Set element offset to 2
ISOURCE             STA Element
ISOURCE             LDA =Value          ! Get the value
ISOURCE             LDB =Array_info
ISOURCE             JSM Get_element

```

UTILITY: Get_bytes

General Procedure: This is similar to the “Get_value” utility. This utility retrieves a substring of a string passed as an argument, having been given the starting byte and the number of bytes to be retrieved.

Special Requirements:

- The storage area set aside for the substring must be large enough to hold all of the substring. This includes not only the string itself, but also two extra words. Remember, a word holds two characters.
- A string must have been passed by the ICALL for the utility to work properly.

Calling Procedure:

1. Store the number of the starting **byte** of the substring desired into the first word of the storage area set aside for the substring. (Note that bytes 0 and 1 are the length word of the string.)
2. Store the number of bytes in the substring into the second word of the storage area.
3. Load register A with the address of the storage area.
4. Load register B with the address of the parameter declaration.
5. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call. The substring is returned starting with the third word of the storage area. (Note: Since the second word contains the length of the substring, you have a string data structure starting with the second word!)

For example —

```

ISOURCE Value:      DAT 2      ! 1st character (ignore length)
ISOURCE             DAT 10     ! Transfer 10 characters
ISOURCE             BSS 5      ! Substring storage area
ISOURCE             SUB
ISOURCE Parameter:  STR
ISOURCE Entry:      LDA =Value  ! Info already stored
ISOURCE             LDB =Parameter
ISOURCE             JSM Get_bytes

```

In this example, Value is the storage area. Since 2 has already been generated and stored in the first word, and 10 in the second, the first 10 bytes of the string would be transferred. Of course, the original string must contain at least 10 characters — or the bytes which are returned may be nonsense. Why was the value 2 stored as the byte number? Because bytes in a string are numbered starting with 0, and bytes 0 and 1 contain the length of the string (see “Data Structures” in Chapter 3).

UTILITY: `Get_elem_bytes`

General Procedure: This is a combination of the “Get_element” and “Get_bytes” utilities. This utility retrieves a substring of an element of a string array passed as an argument. The utility is given the starting byte and the number of bytes to be retrieved.

Special Requirements:

- The storage area set aside for the substring must be large enough to hold all of it. This includes not only the string itself, but also two extra words. Remember, a word holds two characters.
- The array information must be retrieved with the “Get_info” utility before calling this utility.
- The offset of the element in the array must be correct in the array information (word 16 returned by “Get_info”). It should be remembered that the offset of the element is dependent upon the number of dimensions in the array and the length of each. A calculation may be necessary to arrive at the offset when accessing multiple-dimension arrays. The offset is in terms of number of elements.

Calling Procedure:

1. Store the number of the starting byte of the substring desired into the first word of the storage area set aside for the substring. (Note that bytes 0 and 1 are the length word of the string.)
2. Store the number of bytes in the substring into the second word of the storage area.
3. Store the offset within the array information.
4. Load register A with the address of the storage area for the value.
5. Load register B with the address of word 0 of the information returned by the “Get_info” utility (see description of that utility).
6. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call. The substring is returned starting with the third word of the storage area. (Note: since the second word contains the length of the substring, you have a string data structure starting with the second word!)

For example —

```

ISOURCE Value:      DAT 2      ! 1st character (ignore length)
ISOURCE              DAT 10     ! Transfer 10 characters
ISOURCE              BSS 5      ! Substring storage area
ISOURCE Array_info: BSS 30
ISOURCE Element:    EDU Array_info+16 ! Element offset
ISOURCE              SUB
ISOURCE Parameter:  STR (*)
                .
                .
                .
ISOURCE          LDA =Array_info    ! Get array info
ISOURCE          LDB =Parameter
ISOURCE          JSM Get_info
ISOURCE          LDA =2              ! Set offset to 2
ISOURCE          STA Element
                .
                .
                .
ISOURCE          LDA =Value          ! Info already saved
ISOURCE          LDB =Array_info
ISOURCE          JSM Put_elem_bytes

```

In this example, Value is the storage area. Since 2 has already been generated and stored in the first word, and 10 in the second, the first 10 bytes of the string element are transferred. Of course, the string element must contain at least 10 characters — or the bytes which are returned may be nonsense.

Changing the Value of an Argument

At some point during the execution of your assembly language routine, you might want to accomplish the other half of this method of communication with BASIC — namely, changing the value of a BASIC variable which is used as an argument, in effect changing the value of a BASIC variable from the assembly language routine.

As with retrieving a value, there are a number of utilities available for changing a value. The one to use is dependent upon the type of argument passed. The utilities available are —

Name	Used For
Put_value	Simple variables and individual elements of arrays passed as arguments
Put_element	Elements (from arrays passed as arguments)
Put_bytes	Substrings of strings passed as arguments either as simple variables or as individual elements of arrays passed as arguments.
Put_elem_bytes	Substrings of elements (from string arrays passed as arguments)

How each of these utilities is used is described in the immediately following pages.

UTILITY: Put_value

General Procedure: The utility is given the address of the parameter declaration and the address of the value. It changes the value of the BASIC variable associated with the parameter. It works only on simple variables and individual elements of arrays (passed as arguments), of any type.

Special Requirements:

- The value must have the appropriate data structure for the data type of the argument (see “Data Structures” in Chapter 3).
- An actual argument must have been passed by the ICALL for the utility to work properly.

Calling Procedure:

1. Load register A with the address of the storage area of the value.
2. Load register B with the address of the parameter declaration.
3. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call.

Here is an example call to the utility, passing information to an integer argument —

```

ISOURCE Value:      BSS 1
ISOURCE             SUB
ISOURCE Parameter:  INT
                    .
                    .
                    .
ISOURCE             LDA =Value
ISOURCE             LDB =Parameter
ISOURCE             JSM Put_value

```

UTILITY: Put_element

General Procedure: This is similar to the “Put_value” utility. This utility changes the value of a single element in an array passed as an argument. It works on elements of arrays of any type.

Special Requirements:

- The value must have the appropriate data structure for the data type of the argument (see “Data Structures” in Chapter 3).
- The array information must be retrieved with the “Get_info” utility before calling this utility.
- The offset of the element in the array must be correct in the array information for the array (word 16 returned by “Get_info”). It should be remembered that the relative element number of the element is dependent upon the number of dimensions in the array and the length of each. A calculation may be necessary to arrive at the offset when accessing multiple-dimension arrays.

Calling Procedure:

1. Store the element offset into the array information (word 16).
2. Load register A with the address of the storage area for the value.
3. Load register B with the address of word 0 of the information returned by the “Get_info” utility (see description of that utility).
4. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call.

Here is an example call, storing information from Value into element 0 of an integer array —

```

ISOURCE Value:      BSS 1
ISOURCE Array_info: BSS 30
ISOURCE Element:    EQU Array_info+15 ! Element offset
ISOURCE              SUB
ISOURCE Parameter:  INT (*)
ISOURCE              .
ISOURCE              .
ISOURCE              .
ISOURCE              LDA =Array_info    ! Get array info
ISOURCE              LDB =Parameter
ISOURCE              JSM Get_info
ISOURCE              LDA =0              ! Set offset to 0
ISOURCE              STA Element
ISOURCE              .
ISOURCE              .
ISOURCE              .
ISOURCE              STA Value          ! Change the value
ISOURCE              LDA =Value
ISOURCE              LDB =Array_info
ISOURCE              JSM Put_element

```

UTILITY: Put_bytes

General Procedure: This is similar to the “Put_value” utility. This utility changes the value of a substring which is part of a string variable or an individual element of a string array, having been given the starting byte and the number of bytes to be changed as well as the new characters.

Special Requirements:

- The bytes to be transferred are preceded by two words in the storage area. The two words contain the starting byte for the substring and the number of bytes to be transferred.
- A string variable or an element of a string array must have been passed as an argument for the utility to work properly.

Calling Procedure:

1. Store the number of the starting **byte** of the substring to be changed into the first word of the storage area. (Note that bytes 0 and 1 are the length word of the string)
2. Store the number of bytes in the substring into the second word of the storage area.
3. Load register A with the address of the storage area.

4. Load register B with the address of the parameter declaration.
5. Call the utility.

Exit Conditions: There are no error exits from the utility, so it always returns to the instruction following the call.

For example —

```

ISOURCE Value:      DAT 2      ! 1st character (ignore length)
ISOURCE             DAT 10     ! Transfer 10 characters
ISOURCE             BSS 5      ! Substring storage area
ISOURCE             SUB
ISOURCE Parameter:  STR
                    "
                    "
                    "
ISOURCE             LDA =Value   ! Other info already saved
ISOURCE             LDB =Parameter
ISOURCE             JSM Put_bytes

```

In this example, Value is the storage area containing the string to be transferred. Since 2 has already been generated and stored in the first word, and 10 in the second, the first 10 bytes of the string are changed. Why was the value 2 stored as the byte number? Because bytes in a string are numbered starting with 0, and bytes 0 and 1 contain the length of the string (see “Data Structures” in Chapter 3).

UTILITY: Put_elem_bytes

General Procedure: This is a combination of the “Put—element” and “Put—bytes” utilities. This utility changes a substring of an element in a string array which has been passed as an argument. The utility is given the starting byte and the number of bytes to be transferred.

Special Requirements:

- The bytes to be transferred are preceded by two words in the storage area. The two words contain the starting byte for the substring and the number of bytes to be transferred.
- The array information for the array must be retrieved with the “Get_info” utility before calling this utility.
- The offset of the element in the array must be correct in the array information for the array (word 16 returned by “Get_info”). It should be remembered that the offset of the element is dependent upon the number of dimensions in the array and the length of each. A calculation may be necessary to arrive at the offset when accessing multiple-dimension arrays. The offset is in terms of number of elements.

Calling Procedure:

1. Store the number of the starting **byte** of the substring to be changed into the first word of the storage area. (Note that bytes 0 and 1 are the length word of the string.)
2. Store the number of bytes in the substring into the second word of the storage area.
3. Store the element offset into the array information (word 16).
4. Load register A with the address of the storage area for the string to be transferred.
5. Load register B with the address of word 0 of the information returned by the "Get_info" utility (see description of that utility).
6. Call the utility.

Exit Conditions: There are no error exits from the utility. It always returns to the instruction following the call.

For example —

```

ISOURCE Value:      DAT 2      ! 1st character (ignore length)
ISOURCE             DAT 10     ! Transfer 10 characters
ISOURCE             BSS 5      ! Substring storage area
ISOURCE Array_info: BSS 30
ISOURCE Element:    EQU Array_info+16 ! Element offset
ISOURCE             SUB
ISOURCE Parameter:  STR (*)
                *
                *
                *
ISOURCE          LDA =Array_info    ! Get array info
ISOURCE          LDB =Parameter
ISOURCE          JSM Get_info
ISOURCE          LDA =2              ! Set offset to 2
ISOURCE          STA Element
                *
                *
                *
ISOURCE          LDA =Value          ! Info already saved
ISOURCE          LDB =Array_info
ISOURCE          JSM Put_elem_bytes

```

In this example, Value is the storage area for the string to be transferred. Since 2 has already been generated and stored in the first word, and 10 in the second, the first 10 bytes of the string element are changed. It is the responsibility of the software (not shown) to assure that 10 characters of valid data are stored in the remainder of the storage area.

Using Common

Another way to pass information between BASIC and assembly language routines is through BASIC's common area.

You may recall from subprograms in BASIC that if you have a COM statement in the main program, the locations named therein can be accessed by other BASIC subprograms and functions through their own COM statements. Though the subprograms may change the names, the locations are the same. The order of appearance in a COM statement is all-important. If a main program has the statement —

```
COM A, B, C
```

and a subprogram has the statement —

```
COM X, Y, Z
```

then X and A are the same storage location, B and Y are the same, and C and Z are the same.

The same kind of operation is available in your assembly language routines with the COM pseudo-instruction —

```
COM
```

As with the SUB pseudo-instruction, the COM only serves as a preface. It is followed by one or more parameter declarations of the same types as in the SUB —

```
ANY
INT
REL
SHO
STR
```

The FIL is not permitted, since there is no corresponding item within BASIC's COM syntax.

Each pseudo-instruction used after an assembly language COM corresponds to an item in the COM declaration in the main BASIC program. Just as in a BASIC subprogram, the types must agree.¹ However, the ANY pseudo-instruction fulfills the same function here as it does with the SUB pseudo-instruction — to allow any type of item to be passed.

¹ If the types do not correspond, an error results (number 198).

As with SUB, arrays are designated by following the type with an array identifier — (*). If the type is ANY, the array identifier is not allowed.

Each pseudo-instruction reserves three words of memory when assembled. And, like SUB, the words are used to contain a descriptor. The descriptors are used by the variable retrieval utilities for fetching and storing values in the common area. THE SAME UTILITIES USED IN FETCHING AND STORING ARGUMENT VALUES ARE USED FOR THE SAME PURPOSES FOR VALUES IN THE COMMON AREA. These utilities are —

```
Get_info
Get_value
Get_element
Get_bytes
Get_elem_bytes
Put_value
Put_element
Put_bytes
Put_elem_bytes
```

The utilities are called in the same fashion and are subject to the same restrictions. See the description of the utilities in the preceding sections of this chapter to determine how they are used.

The item pseudo-instructions used with the COM pseudo-instruction can have their own labels, just as the parameter declarations used with a SUB may have. And just as in a BASIC subprogram, they need not have the same names as were given the corresponding items in BASIC. For example, suppose the following BASIC common statement exists at the time of a call to an assembly language routine —

```
COM Q(20),Z$(10)
```

then you could access Q(*) and Z\$ by using these pseudo-instructions —

```
ISOURCE      COM
ISOURCE X:    REL (*)
ISOURCE Y:    STR
```

Note the differences in names.

If the number of item pseudo-instructions in the assembly language routine exceeds the number of items in common at the time the routine is called, an error results (number 199).

A COM pseudo-instruction sequence need only be set up once per module. Each routine within the module has access to the information within the sequence. The three-word descriptors are filled, and type-checking occurs, only once — at the first ICALL of a routine within the module.

Busy Bits

Overlapped processing in the 9835A/B is partially implemented through the facility of “busy bits”.

Each variable located in the BASIC value or common areas has associated with it two bits which are independent of the value — a “read” busy bit, and a “write” busy bit. Each time an I/O operation is executed that cannot be buffered, one of the busy bits is set. If a variable is having its value changed by the I/O operation, then the read busy bit is set. If the variable is outputting its value in the I/O operation, then its write busy bit is set. If a variable is not involved in a pending I/O operation both bits are cleared. When the I/O operation is completed, the busy bits for the variables involved are cleared.

When an I/O operation is encountered during execution of BASIC statements, the appropriate busy bits are set and a request is made by the operating system for the resources to satisfy the operation. Until that operation is complete, BASIC (in OVERLAP mode), continues to execute succeeding lines in the program until it encounters a statement which contains variables with busy bits that are set.

If the statement is attempting to use the value of a variable and its read busy bit is set, then the further execution of the statement waits until the busy bit is cleared. The same is true for a statement attempting to change the value of a variable when either its read or write busy bit is set. When the I/O operation completes, the busy bits are cleared and the waiting statement is executed.

In short, overlapped processing uses busy bits as a signal as to whether a statement can be executed or not.

If an ICALL statement is executed with overlapped processing, it is possible that a BASIC variable in the common area may be “busy” when the routine wants to access it. (The busy bits of variables passed as arguments are checked — and are non-busy — before the ICALL is executed.) Although it is still possible to access the variable without regard to the status of the busy bits, frequently that is not a desirable programming approach. You may on occasion want to check the value of the busy bits when you suspect the user of the routine may be using overlapped processing.

Busy bits are checked from an assembly program using the “Busy” utility to be described shortly. If you are checking the bits for a busy condition, and the busy condition is set, it remains set throughout the time you are in the assembly routine. For it to become un-busy, you must exit the routine and permit the operating system a chance to perform the I/O operation and clear the busy bits.

For example —

```
330 ICALL Sort(Busy)
340 IF Busy THEN 330
```

If the Sort routine exits, setting Busy to 0 if a busy condition is not encountered, and to non-zero otherwise, this is a tight loop which keeps trying to execute Sort until the common variables which are busy become un-busy and it can proceed on its way. By exiting the routine after each unsuccessful attempt, the operating system is given an opportunity to perform the I/O operation which has the variable(s) tied up.

UTILITY: Busy

The Busy utility checks the status of the busy bits of a variable in BASIC’s common area. It is not necessary to check the busy bits of a variable passed as an argument since all arguments are checked upon calling a routine (and the call is executed only when all the arguments are not busy).

General Procedure: The utility is given the location of the common declaration for the variable. It returns the value of the busy bits for that variable into the A register.

Special Requirements: This utility should only be used for variables in common.

Calling Procedure:

1. Load register B with the address of the pseudo-instruction of the common declaration to be checked.
2. Call the utility.

Exit Conditions: The utility returns the busy bits in the A register. The “read” busy bit is in bit 0 and the “write” busy bit is in bit 1. The other bits are not disturbed.

In the following example, if any of the busy bits among three common variables is set, a flag is set and the routine is exited —

```

ISOURCE      COM
ISOURCE Variable1: INT
ISOURCE Variable2: SH0
ISOURCE Variable3: REL
      *
      *
      *
ISOURCE      SUB
ISOURCE Busy_bits: INT
ISOURCE Sort: LDB =Variable1
ISOURCE      JSM Busy
ISOURCE      SAL 2
ISOURCE      LDB =Variable2
ISOURCE      JSM Busy
ISOURCE      SAL 2
ISOURCE      LDB =Variable3
ISOURCE      SZA **4
ISOURCE      LDA ==1
ISOURCE      LDB Busy_bits
ISOURCE      JSM Put_value
ISOURCE Work: ! Continue processing

```


Chapter 7

Table of Contents

I/O Handling

Peripheral-Processor Communication	133
Interfaces	134
Registers	134
Select Codes	134
Status and Control Registers	136
Status and Flag Lines	137
Programmed I/O	138
Interrupt I/O	138
Priorities	140
Interrupt Service Routines and Linkage	140
Access	141
Utility: Isr_access	143
State Preservation and Restoration	145
Indirect Addressing in ISRs	146
Direct Memory Access (DMA)	147
DMA Registers	148
DMA Transfers	149
BASIC Branching on Interrupts	150
ON INT Statement	150
Signalling	151
Additional Pre-Defined Symbols	153
Prioritizing ON INT Branches	153
Environmental Considerations	155
Disabling ON INT Branching	156
Mass Storage Activities	156
Reading from Mass Storage	157
Utility: Mm_read_start	158
Utility: Mm_read_xfer	159
Writing to Mass Storage	160
Utility: Mm_write_start	161
Utility: Mm_write_test	161
System File Information	163
Utility: Get_file_info	164
Utility: Put_file_info	165
Printing	166
Utility: Printer_select	166
Utility: Print_string	167

Chapter 7

I/O Handling

Summary: This chapter describes the various techniques of handling the receiving and sending of information to peripheral devices. Topics are: a review of I/O machine instructions, registers, applicable utilities, interrupts and interrupt service routines, handshake I/O, direct memory access, and mass storage devices.

A major usage for assembly language programs is to improve or customize the performance of the 9835A/B with respect to data transfers with peripheral devices. The types of devices dealt with are those which communicate via the various interface cards (e.g., HP98032, HPIB, etc.). The types of I/O which the assembly language supports are **programmed** (handshake-type), **interrupt**, and **direct memory access** (or DMA).

A number of detailed examples have been provided demonstrating the various types of I/O on different interfaces. These examples can be found in Appendix H.

Peripheral-Processor Communication

All I/O, except for that to the internal devices (tape cartridge, keyboard, printer, CRT or SLD), necessarily takes place through the "backplane". The backplane is that physical area of the machine where the interface cards are inserted (also known as the I/O "slots").

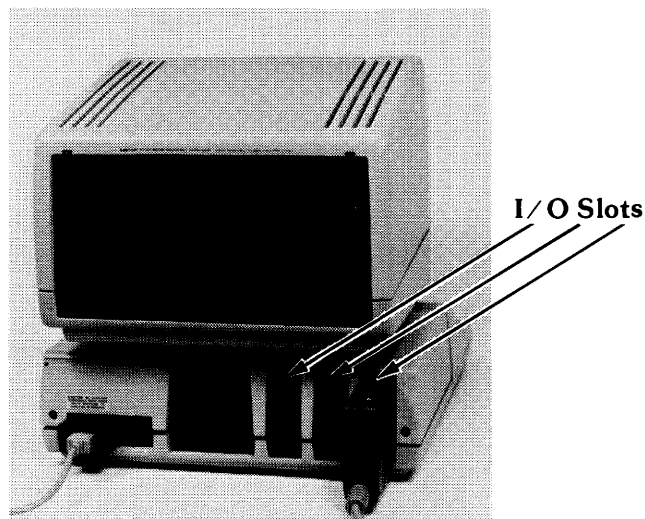


Figure 8. Location of I/O Slots (Backplane)

The backplane serves as an intermediary between the processor and the peripheral interfaces. The internal addressing of the backplane is transparent, both to the interfaces and to the programmer.

Interfaces

The processor does all its talking, through the backplane, to peripheral interfaces, never directly to a peripheral itself. An interface is a complex electronic circuit which provides mechanical, electrical, data format, and timing compatibility between the 9835A/B and the peripheral device to which it is connected. From a programmer's point of view, the primary task of an interface is to provide a means of exchanging data between the 9835A/B and the peripheral. A well-designed interface isolates the programmer from the details of electronics and timing, appearing as a simple "black box" through which information is exchanged.

The processor can talk to as many as 14 peripheral interfaces through the backplane. Each can be talked to individually, and there may be a mix of peripherals using programmed, interrupt, or DMA types of transfers.

Individual I/O operations (i.e., exchanges of single words) occur between the processor and one interface at a time, although interrupt and DMA modes of operation can be programmed to allow automatic interleaving of individual operations.

A peripheral is addressed through a select code and a transfer occurs through four special registers reserved for the purpose. These will each be discussed shortly.

Discussion of the techniques and methods presented in this chapter uses the common HP interfaces as examples. A full discussion of the operation of these interfaces can be found in the *Interfacing Concepts* manual (HP part number 09825-90060) and also from your Sales and Service office (list in Appendix K).

Example programs utilizing various I/O techniques with a number of the standard interfaces can be found in Appendix H.

Registers

All I/O operations go through a set of four registers maintained by the 9835A/B. The four — R4, R5, R6, and R7 — are the sole means of communicating data between the processor and peripheral interfaces. While the registers are actually on the interface cards, they may be thought of as being in the computer memory. This makes the cards themselves accessible by simple memory referencing instructions.

The 9835A/B sees the registers as single-words and always sends or receives a full word of data when it references one of them. If a particular interface utilizes less than the full sixteen bits (when exchanging 8-bit extended ASCII data bytes, for example), then the most significant bits (8 through 15) are received as zeroes. On output, if fewer than 16 bits are utilized by the interface, it ignores the most significant bits. The value of these bits, in this case, is a “don’t care” (i.e., may be any pattern of ones or zeroes).

All of the HP 98030 series of interface cards use the registers as follows —

Register	On Input	On Output
R4	Primary Data In	Primary Data Out
R5	Primary Status In	Primary Status Out
R6	Secondary Data In	Secondary Data Out
R7	Secondary Status In	Secondary Status Out

The R4 register, then, is almost always used for data transfers. R5 is always used for status and control information. The “secondary” registers — R6 and R7 — perform the indicated functions only nominally. The exact interpretation as to how the register is used depends upon the interface card being used (see the Interfacing Concepts manual for details).

In order to give some specific examples for using the registers, the 98032 16-Bit Parallel Interface (sometimes called General Purpose Input/Output — GPIO) is used. This card defines the secondary registers as —

Register	On Input	On Output
R6	High-Byte Data In	High-Byte Data Out
R7	(unused)	Trigger

Select Codes

As mentioned earlier, more than one interface card may be connected to the 9835A/B. It becomes necessary, then, that there be a mechanism whereby a particular interface can be chosen to respond when an I/O register is referenced for either input or output. This mechanism is the Peripheral Address Register (Pa).

Pa holds a binary number in the range 0 to 15 (utilizing only the lower four bits of the word, 0 to 3). Each interface has an externally-settable select code switch which can also be set to a value between 0 and 15. However, since select codes 0 and 15 are reserved for the internal printer and tape cartridge unit, respectively, the permissible select code settings are 1 through 14.

Whenever an operation to one of the I/O registers is performed, the 9835A/B makes the contents of the Pa register available to all the interfaces connected to the backplane. Each card compares the value with its own select code. If they match, the interface responds to the operation.

So, for example, if the following statements are executed in turn —

```
ISOURCE   LDA #8      ! Choose peripheral on select code 8
ISOURCE   STA Pa
ISOURCE   LDA R4       ! Read from the interface
```

then a word of data is read from the interface card set to select code 8. (The data was read in the third line; this is discussed in “Programmed I/O” below.)

The label “Pa” is reserved by the assembler for the Peripheral Address register.

Status and Control Registers

The primary purpose of any interface is to allow data to be exchanged between the computer and the peripheral device to which it is connected. But HP’s 98030 series of interface cards are even more versatile, possessing a programmable capability of their own. This in turn provides optional capabilities with the card that can be set and changed by control instructions from the 9835A/B. (For details on what capabilities are provided, consult the Interfacing Concepts manual.)

The programming of the interface is done by the 9835A/B using the R5 register. Some of the interfaces use other registers for extended control bits (these are also described in the Interfacing Concepts manual).

Interface cards can also return information to the 9835A/B about which optional programming features are currently selected. This information, called the status byte, is obtained through an input operation using register R5. The status byte (8 bits) is determined solely by the characteristics of the interface card being addressed in the Pa register. Again, information on particular cards can be found in Interfacing Concepts).

Remembering that these registers are not really memory locations, but instead are registers on the card being addressed by the Pa register, storing information to these locations is not the same as storing to other memory locations or registers. For example, storing a value in R5 to set the control register sends the information to the addressed interface. Later, if you were to read a value from R5, the information you sent would not be what is returned. Instead, the contents of the status register in the interface would be returned.

Status and Flag Lines

Whenever an I/O register is accessed, the interface with the same select code as is in the Pa register responds. The primary response depends upon the nature of the interface and which register is accessed (see discussion above). However, in all cases there is a secondary effect. Part of every interface's response is to set or clear the Status and Flag lines.

The **Status** line (not to be confused with the status register discussed above), is a single bit indicating whether the interface is operational or not. By inclusion, this can also mean the status of the actual peripheral to which the interface is connected. For example, if a peripheral device has a line coming from it that indicates its power is on, it could be connected to the Status line in the interface. Then the program could quickly determine whether the device is turned on or off. As another example, a printer might have the Status line connected to the out-of-paper indicator (should it have one) to indicate to the program when it is inoperable because of lack of paper.

The **Flag** line is a momentary "busy/ready" indicator used to keep the computer from getting ahead of the peripheral. The line shows that the interface is busy processing the last task given it by the 9835A/B or that it is ready for another operation. If the line is set, it indicates "ready"; if the line is cleared, it indicates "busy". For example, if the computer has a sequence of ASCII characters to send to a slow printer, it sends one character (making the Flag line "busy") and then waits for the Flag line to go "ready" again before sending the next character.

There are four instructions, part of the I/O group, which can check these lines —

SFS	Skip if Flag line is set (i.e., "ready")
SFC	Skip if Flag line is cleared (i.e., "busy")
SSS	Skip if Status is set (i.e., "operational")
SSC	Skip if Status is cleared (i.e., "non-operational")

These instructions have the capability of skipping up to 31 locations in a forward branch, up to 32 locations in a backward branch, or to the same instruction.

Programmed I/O

Programmed I/O is the process whereby software controls the transfer of information between memory and an interface. In the process the program must decide when and where to make the transfer, how to make it, and how much information to transfer. The decision even to originate the transfer comes under program control.

The Status line can be used to determine the availability of an interface. The interface is selected, under program control, by the contents of the Pa register. Then the Status line is checked to see if the interface (and by inclusion its associated peripheral) is operational.

After an operational interface has been chosen, the Flag line can be used to determine when the interface (i.e., peripheral) is ready for a transfer and when it has not finished with the previous transfer.

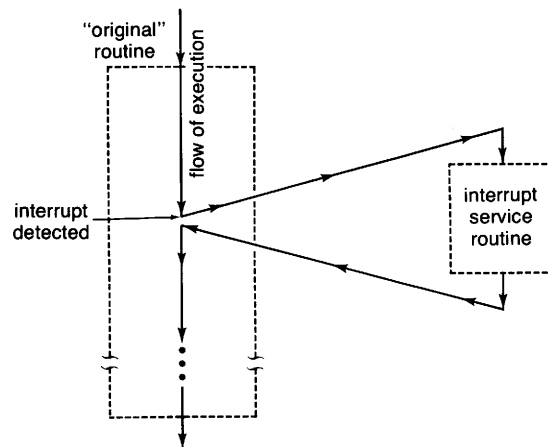
With sufficient checks of Flag and Status before and between I/O operations, it is possible to eliminate initiating an I/O operation to an interface which isn't ready for it. For example, a simple output driver for the 98032 interface is —

```
ISOURCE  STA Pa          ! Choose the peripheral
ISOURCE  SSS #+3         ! Check for operational device
ISOURCE  LDA #164        ! Not operational, error number 164
ISOURCE  JSM Error_exit  ! Inform user
ISOURCE  SFC *           ! Wait until last operation is done
ISOURCE  STB R4          ! Output the next word to the interface
ISOURCE  STB R7          ! Start the handshake
```

Interrupt I/O

Interrupt I/O is a means of allowing control to pass temporarily to an assembly language routine other than the routine (BASIC or assembly language) currently executing. The “interrupt”, which causes the control to be passed, is detected through the backplane and is associated with a particular interface. After the “interrupt service” routine completes its tasks, control is passed back to the original routine.

The process looks something like this —



The sequence of events in interrupt I/O can be detailed as follows —

1. The interface sends a request for service to the backplane which passes it along to the processor.
2. The processor alters the flow of execution so that the routine associated with that interrupting source can be executed. The processor saves its place in the interrupted routine so that it can later return to it.
3. The interrupt service routine is executed, performing whatever functions are desired. Frequently these functions involve some form of programmed I/O or direct memory access. The service routine may signal an end-of-line BASIC branch, indicating to BASIC that some condition occurred (discussed below).
4. The service routine returns the processor to the interrupted routine so that the “original” process can resume.

The uses for interrupt I/O are so diverse that it is difficult to generalize about them. However, one particular use is fairly well-defined and of general applicability — data transfers.

Interrupt I/O is normally used in data transfers whenever a particular data device has a transfer rate which is significantly slower than that of the computer. Since the 9835A/B has a transfer rate of around 10 000 characters per second, peripheral devices with transfer rates slower than this number are candidates for interrupt I/O.

The usual approach is to transfer a word to or from the peripheral device, then go away to do some other processing while waiting for the device to interrupt by becoming “ready” for another transfer.

Priorities

Select codes are assigned hardware “priority” levels to control what should be processed when an interrupt service routine is executing and another interrupt is received, or when two or more simultaneous interrupts are received.

There are two priority levels —

High	for select codes 8 to 15
Low	for select codes 0 to 7

An interrupt received from a high-priority select code may interrupt a service routine which is executing for an interrupt from a low-priority select code. But an interrupt from a low-priority select code may not interrupt any other service routine.

Interrupt Service Routines and Linkage

An interrupt service routine is associated, or “linked”, with a select code by the `Isr_access` utility described later. This linkage establishes where the interrupt service routine resides, and to which select code it applies.

An interrupt service routine may be placed anywhere in the ICOM region. The routine typically does one or more of the following —

- Talks to the interface (i.e., satisfies or acknowledges the interface’s interrupt).
- Passes data to (or retrieves data from) the rest of the program, when appropriate.
- Breaks the linkage, if desired.

The method of talking to the interface depends upon the type of interface. Some devices or applications do not require the passage of data; the acknowledgement of the interrupt is usually the desired effect in such cases.

The linkage can be “broken” (or terminated) during an interrupt service routine by executing one of two statements. If the linked select code is high-priority, the statement is —

```
JSM End_isr_high,I
```

If the linked select code is low-priority, the statement is —

```
JSM End_isr_low,I
```

The service routine is exited with a RET 1 instruction.

Here is an example of a short interrupt service routine which simply reads a word of data from the interface —

```
ISOURCE  Lf:          EQU 10
          .
          .
          .
ISOURCE  Isr:          LDA R4          ! Retrieve character from interface
ISOURCE                      CPA #Lf      ! Is it a line feed?
ISOURCE                      JMP Terminate ! Yes, so jump
ISOURCE                      STA R7       ! No, so trigger another handshake
ISOURCE                      RET 1
ISOURCE  Terminate:    JSM End_isr_low,I
ISOURCE                      RET 1
```

NOTE

Utilities cannot be called from an interrupt service routine.
Attempts to do so may lock up the machine.

Access

The operating system (OS) contains a mechanism to regulate requests for hardware capabilities in order to eliminate conflicting uses of these capabilities. For instance, since there is only one DMA channel, it is necessary that there be a mechanism to prohibit two simultaneous DMA transfers.

The OS mechanism which regulates the use of DMA (and also interrupt) transfers either grants or does not grant what is called “access”. Before starting either an interrupt or DMA operation, access should be requested from the operating system.

Another example — suppose a device operating on a high priority select code has a relatively slow data rate. This is an ideal situation in which to use interrupt driven I/O. Suppose further that the device operates in such a fashion that the data must be transferred within a fixed time period following its issuance of an interrupt or the data is lost (the internal tape drive is such a device.) If there are other interrupt type transfers operating concurrently on other high priority select codes, it may not be possible to service our slow device within the necessary time frame. When the operating system grants access, this type of conflict is impossible.

Users of the assembly language system are required to request access from the operating system. The OS grants access if granting this access does not compromise any previously granted access.

Devices such as that discussed above which require interrupt service within a specified time frame are called “synchronous”, and need “synchronous” access. Devices with no such time constraints are called “asynchronous”, and need “asynchronous” access.

The regulation of access incorporates the following points —

- When the operating system grants synchronous access to an operation, it is guaranteeing that the requesting process will have its interrupts serviced with maximum priority.
- DMA conflicts with synchronous access since DMA’s cycle stealing causes the processor to run slower and could thus compromise a synchronous process.
- Synchronous access on a low priority select code in conjunction with asynchronous access on a high priority select code is conflicting since the asynchronous device could interrupt the synchronous ISR, thus compromising the timing requirements of the synchronous device.
- Synchronous and asynchronous access on the same priority level is also conflicting. Remember an interrupt request on the same priority level as a currently executing ISR will not be processed until the executing ISR completes.

The following table summarizes the granting of access —

		Access Already Granted								
		Abortive		ASYN		DMA	SYN			
		L	H	L	H		H	L		
Access Requested	Abortive	{	Low							d
			High						d	d
	ASYN	{	Low							x
			High						x	x
	DMA						x	x	x	
	SYN	{	High		d		x	x	x	x
			Low	d	d	x	x	x	x	x

blank = Granted

x = Not granted



d = Dangerous, but granted

BASIC statements also obtain and release access as I/O is performed. The following table lists some of the ways access is used by the system:

Use	Access
Cartridge Operations	SYNC (HIGH select code)
Flexible Disk Operations	DMA
PRINT, PRINT USING	ASYNC
Plotter Drivers	ASYNC
CARD ENABLE	ASYNC
ENTER/OUTPUT INT	ASYNC
ENTER/OUTPUT DMA	DMA
ENTER/OUTPUT FHS ¹	DMA

In general, single BASIC statements could cause access to be granted and released several times. For example, the cartridge operations obtain and release synchronous access once for each physical record transferred.

UTILITY: `Isr_access`

This utility is used to request access and, if the access is granted, to create the linkage between an interrupt service routine (ISR) and a select code. Pressing RESET ( ) during execution of the utility may cause a SCRATCH A to be issued.

General Procedure: The utility is told where the ISR resides and what kind of access is required. If access is granted, it returns successfully. If access is not granted immediately, it keeps trying periodically until it is successful or until a specified number of attempts have been made (in which case it returns unsuccessfully).

Special Requirements: The B register must contain information as follows —

Bits	Description
0–3	Select code to be linked to the ISR
4–5	Access code (see next page)
8–14	Number of attempts to be made before aborting

¹ In addition to obtaining DMA access (which in this case is used just to ensure there is no synchronous access granted), the FHS drivers disable all interrupts during the actual transfer loop.

The access codes are —

- | | |
|---|---|
| 0 | Abortive access |
| 1 | Asynchronous access with programmed I/O |
| 2 | Asynchronous access with DMA |
| 3 | Synchronous access with programmed I/O |

Calling Procedure:

1. Load register A with the address of the ISR.
2. Load register B with the information described above.
3. Call the utility.

Exit Conditions:

RET 2 If the attempt at linkage is successful, the utility returns to the second word following its call. Register Pa is set to the select code; if access code 2 was specified then Dmapa has also been set to the select code.

RET 1 If the attempt at linkage is unsuccessful, the utility returns to the first word following the call. Register A contains an indication of the type of difficulty encountered —

- 1 Access couldn't be obtained after specified number of attempts.
- 2 Select code is still linked to an assembly language ISR.

Note: Access code 0 (abortive access) should be used with caution. An interrupt routine with abortive access can exist on the same priority level as an interrupt routine with synchronous access. If the abortive routine is in progress when an interrupt occurs requiring the synchronous service routine, the abortive routine will finish before the synchronous routine can be serviced. The timing requirements of the synchronous routine might thus be violated.

Access code 0 is intended to be used by routines that will be executed only extremely infrequently. For instance, if the 9835A/B is monitoring a potentially dangerous manufacturing process, it may be necessary to have an interrupt service routine to shut down the process when something goes awry. This could be accomplished with an abortive routine. The advantage (and also the reason for the previously mentioned caution) of access code 0 is that no other modes of access are prohibited by its use. Thus, the infrequently used routine will not prevent another routine from getting the type of access it needs.

As an example of the use of the `Isr_access` utility, suppose the ISR from page 141 is to be linked to select code 2 for asynchronous access. The following would be a sequence to establish such a linkage —

```

ISOURCE      EXT Isr_access
               .
               .
               .
ISOURCE      LDA =Read
ISOURCE      LDB =(64*256)+(1*16)+2  ! 64 trials, asynch, SC 2
ISOURCE      JSM Isr_access
ISOURCE      JMP Error
               .
               .
               .
ISOURCE Error: ISZ A
ISOURCE      JMP Nested_isr      ! Handler for SC busy
ISOURCE      JMP No_access       ! Handler for time-out
               .
               .
               .

```

State Preservation and Restoration

When an interrupt is detected and an interrupt service routine is called, the processor automatically saves the state of some of the registers so that their values can be restored upon return from the ISR. Other registers are left alone and if your service routine uses them, it is up to your ISR to save them and restore them before returning from the ISR.

The registers which are automatically preserved are —

A
B
C
Cb
P
Pa

Also, the state of the Overflow and Extend processor flags are preserved and restored before the return from the interrupt.

If your ISR contains any of the following types of instructions —

Indirect addressing
Stack group
CLR
XFR

and the operand of the instruction(s) is an address in the ICOM region, then it is necessary that the following instruction sequence be executed in the ISR before any such instruction is executed —

```
Save35: BSS 1
        .
        .
        .
        LDA 35B
        STA Save35
        LDA 34B
        STA 35B
        .
        .
        .
```

Then, before the ISR exits, and after the affected instructions have been executed, the following sequence must be executed —

```
        .
        .
        .
        LDA Save35
        STA 35B
        .
        .
        .
```

Indirect Addressing in ISRs

Indirect addressing in ISRs can produce anomalies unless the following rules are followed —

1. If indirect addressing is employed with the operand being an address in the ICOM region, one of the processor registers must be preserved. For the method of doing this, consult the “State Preservation and Restoration” section immediately above.

2. If indirect addressing is used in a JMP or JSM (including any jumps to external symbols or symbols more than 512 words away from the current instruction, both of which have implied indirect addressing), then the most significant bit must be set in the address. For example, instead of —

```
EXT Sub
  *
  *
  *
JSM Sub
```

in an ISR the procedure must be —

```
EXT Sub
  *
  *
  *
JSM (=Sub+100000B);I
```

Direct Memory Access (DMA)

Direct memory access (DMA) is a means to exchange entire blocks of data between memory and peripherals. A block is a series of consecutive memory locations. Once started, the process is automatic; it is done under processor control, regulated by the interface. Since only the 98032A interface supports DMA, the following discussion is in terms of that interface.

To the peripheral, the DMA operation appears as programmed I/O. The transfer, however, is actually performed by special DMA hardware. Information regarding the transfer is stored in the DMA registers for the DMA hardware to use. This information is the select code, the initial memory location, and the number of words to be transferred. The memory location register and the count register are successively adjusted after each word transferred until the transfer is complete. Upon completion of the transfer, the interface and the DMA hardware stop automatically.

The direction of the transfer is specified before the transfer takes place. It can be specified as either “inward” (i.e., from the peripheral to memory), or “outward” (i.e., from the memory to the peripheral). To set the direction outwards, the instruction —

SDO

is used. To set the direction inwards, the instruction —

SDI

is used.

DMA Registers

There are three registers which contain information used by the DMA hardware — Dmapa, Dmama, and Dmac. Before any DMA transfer takes place, the appropriate values must be loaded into these registers.

Dmapa contains the peripheral address of the device requesting DMA. Only the least significant bits of the register specify the select code which is to be the peripheral side of the DMA activity. During DMA transfers, the address bus takes its address from the Dmapa register rather than Pa as in other I/O transfers. The value is supplied to Dmapa by the `Isr_access` utility when it grants DMA access.

Dmama contains the address of the first word in memory (i.e., lowest address) where the data transferred is (or will be) stored. After each word transferred, this register is automatically incremented.

Dmac is the count register for a DMA transfer. Before the transfer begins, it should be set to $n-1$, where n is the number of words to be transferred. After each word transfer, the count is decremented. If, during a word transfer, the value of Dmac is 0 (meaning that this is the last word to be transferred), the processor automatically informs the interface that the DMA operation will be complete after the present word is transferred. In the case of inputs where the amount of transferring data is unknown in advance, Dmac should be set to a very large number in order to disable the signal to the interface.

DMA Transfers

DMA transfers are accomplished with six distinct actions.

First, the `Isr_access` utility is used to obtain access to the DMA channel and to set up the ISR linkage used when the transfer terminates.

Second, the direction is set using an SDO or SDI instruction. If no direction is set, then any previous setting of the direction prevails.

Third, the appropriate values are stored into the DMA registers.

Fourth, the DMA requests are enabled using the instruction —

```
DMA
```

Fifth, a “Start DMA” command is given to the interface using programmed I/O. With the 98032 interface, this command is the value 320₈ using the Primary Control register (R5-Out).

Finally, when the DMA transfer is complete, the interface generates an interrupt which causes the processor to branch to the designated ISR. This ISR should disable the card, and then disable the DMA mode with the instruction —

```
DDR
```

The following is part of an ISR which demonstrates a typical set-up for a DMA inward transfer (in this case 1K words placed into a buffer in memory) —

```
ISOURCE      LDA #1023
ISOURCE      STA Dmac      ! Set up DMA count
ISOURCE      LDA #Buffer
ISOURCE      STA Dmana     ! Establish address of receiving area
ISOURCE      SDI           ! Set DMA as inwards
ISOURCE      SFC *        ! Wait for Flag line to clear
ISOURCE      LDA R4
ISOURCE      STA R7       ! Start input handshake
ISOURCE      DMA          ! Enable DMA request
ISOURCE      LDA #320B    ! "Start DMA" command
ISOURCE      STA R5
```


BASIC Branching on Interrupts

The handling of interrupts can be integrated into BASIC programs by using the ON INT statements. The object is to allow the flexibility of combining the high-level features of BASIC with the capabilities of assembly language in asynchronous I/O applications. And since ISRs cannot use the system utilities, in particular those that access a BASIC variable, a means of taking action on an interrupt after completion of the ISR is a necessity.

ON INT Statement

The ON INT statement is an executable BASIC statement which acts in a similar fashion to the ON KEY statement (see the 9835A/B Operating and Programming Manual). The statement allows the BASIC programmer to specify where, in his BASIC program, to branch whenever an interrupt is signalled for the select code he specifies.

As with the ON KEY statement, there are three ways these branches can be taken —

```
ON INT # {select code} [, {priority} ] CALL {subprogram name}
ON INT # {select code} [, {priority} ] GOSUB {line identifier}
ON INT # {select code} [, {priority} ] GOTO {line identifier}
```

Whenever an interrupt is signalled from an ISR for a particular select code, if ON INT has been executed for that select code, then at the end of execution of the BASIC line which was executing when the signal came, the indicated branch in the ON INT is taken.

In the GOTO version, the branch is “absolute”, which is to say that the program goes to the line indicated and picks up its execution there, forgetting where it was before. This has the effect of an “abortive” type of branch, and should only be used by the BASIC programmer when he wants the program to resume execution at some pre-determined point after handling an interrupt, without regard to where the program was before the interrupt occurred.

In the CALL and GOSUB versions, the branch is only temporary. After the subprogram or subroutine has been executed and the SUBEXIT, SUBEND, or RETURN (as appropriate) has been executed, then the program returns to the line following the one where it was interrupted. This is the same as if the CALL or GOSUB was in between the interrupted line and the one following it.

The {line identifier} and {subprogram name} in the CALL, GOSUB, and GOTO statements are the same as elsewhere in BASIC, except that a CALL may not have any parameters.

Chapter 8

Table of Contents

Debugging

Stepping Through Programs	170
Individual Instruction Execution	170
Setting Break Points	174
Simple Pausing	174
Transfers	175
Environments	176
Data Locations	177
IBREAK Everywhere	178
Number of Break Points	179
Clearing Break Points	179
Interrogating Processor Bits	180
Protected Memory	180
Dumps	181
Value Checking	183
Functions	184
DECIMAL	184
OCTAL	184
IADR	185
IMEM	186
Patching	187

The {select code} specified in an ON INT statement restricts the branching action to occurring only when the assembly language triggers the ON INT condition for that select code. The interrupt may have occurred in actuality on another select code. This can be a way of allowing more than one branch for interrupts from a single interrupting device.

As an example —

```
100 ON INT#3 GOSUB Print_result
110 ON INT#5 GOSUB End_data
```

Should anywhere in the program an interrupt occur, causing an assembly language interrupt service routine to be executed, that assembly language ISR has the capability to cause either the branch of line 100 or the branch of line 110 to be taken. Thus, an assembly language ISR signals BASIC either to print an intermediate result or to note that all data has been processed.

Signalling

The {select code} specified in an ON INT statement restricts the branching action to occurring only when an interrupt is “signalled” for that select code. In actuality, an interrupt may not have occurred on that select code at all. Conversely, an interrupt may occur on the select code, but BASIC and its ON INT condition may never hear about it. It is necessary for the ISR which does the actual handling of an interrupt to inform, or “signal”, the operating system that the interrupt occurred and trigger the ON INT conditions which may be set up at the time.

The responsibility of the ISR to signal the ON INT is also an opportunity. This signalling allows you in an ISR to decide whether or not you want BASIC to know about the interrupt. If you do not want BASIC to know, simply do not signal the condition. The signalling also allows you to signal different interrupt conditions. An example of doing this might be a case where, after an interrupt, a peripheral indicates whether it wants to input or output data. Your routine could signal one select code to execute an input routine and signal another select code to execute an output routine.

To signal an ON INT, your ISR must execute the following instructions —

```
ISOURCE      LDB Isr_psw
ISOURCE      LDA #103B
ISOURCE      STA B,I
ISOURCE      ADB=3
ISOURCE      LDA Mask      ! Determines which SC to signal
ISOURCE      STA B,I
ISOURCE      STA Isr_flag,I
```

Mask necessarily contains the select code to be signalled. Rather than containing the number of the select code, however, it has the bit set for the appropriate select code. For example, if you are signalling select code 2, you set bit 2 to 1 in Mask and leave the others 0. Similarly, if you are signalling select code 5, you set bit 5. Thus, the statement containing Mask in the above could just as easily be a literal. For example —

```
LDA =32
```

would signal select code 5.

When you want to signal a select code after others have already been signalled, a slightly different instruction sequence is required —

```
ISOURCE      LDB Isr_psw
ISOURCE      LDA =103B
ISOURCE      STA B,I
ISOURCE      ADB =3
ISOURCE      LDA Mask
ISOURCE      DIR
ISOURCE      IOR B,I      ! Ors in the select code
ISOURCE      STA B,I
ISOURCE      EIR
ISOURCE      STA Isr_flag,I
```

Mask is the same as above.

As a further example, suppose you want both to signal BASIC when a device sends a line-feed character to the computer, and to terminate the ISR's linkage. Then the ISR might appear as —

```
ISOURCE  Lf:      EQU 10
          .
          .
          .
ISOURCE  Isr:      LDA R4
ISOURCE            CPA =Lf      ! Is it a line feed?
ISOURCE            JMP Terminate ! Yes, so jump
ISOURCE            STA R7      ! No, so trigger another handshake
ISOURCE            RET 1
ISOURCE  Terminate: JSM End_isr_low,I
ISOURCE            LDB Isr_psw  ! Signal BASIC
ISOURCE            LDA =103B
ISOURCE            STA B,I
ISOURCE            ADB =3
ISOURCE            LDA =1      ! ON INT #1 signalled
ISOURCE            STA B,I
ISOURCE            STA Isr_flag,I
ISOURCE            RET 1
```

Additional Pre-Defined Symbols

`Isr_flag` and `Isr_psw` are pre-defined symbols in the assembler. Also pre-defined are two other symbols used by the assembler — `End_isr_low` and `End_isr_high`. These symbols may not be redefined.

Prioritizing ON INT Branches

Since more than one interrupt may occur while a single BASIC statement is executing, it is possible that by the time the line finishes there may be a number of ON INT branches waiting to be executed. In such situations you may want to assure that some ON INT branches are taken before others, or that you finish one routine (caused by an ON INT GOSUB or ON INT CALL) before you start another. This can be achieved by using the {priority} option of the ON INT statement, thereby “prioritizing” the branching caused by interrupts.¹

There is a “system priority” for ordering this interrupt branching. For an ON INT to be honored at the end of a BASIC line, its priority must be greater than the current system priority.

Initially, the system priority is set to 0. When a BASIC line finishes, and there is at least one ON INT branch pending which is greater than the system priority, then the system takes the branch associated with the ON INT with the greatest {priority}. The values assigned to {priority} may be any integer numeric expression from 1 to 15. If {priority} is omitted, 1 is assumed.

If the ON INT branch to be executed is a GOTO, then the system priority level is unchanged. But if the branch to be executed is a GOSUB or a CALL, then the system priority level is changed to the priority level of the ON INT. Whenever the subroutine or subprogram is finished executing, then the previous system priority level is restored.

Thus, with the GOSUB and CALL versions, there are two effects involving priorities —

- The subroutine or subprogram is not allowed to execute until its priority is the highest one pending.
- Whenever the subroutine or subprogram is executing, it locks out any other interrupting branches unless they have a higher priority.

¹ This “prioritizing” also holds between the various types of end-of-line branch statements that have the priority parameter. Thus an ON KEY with high priority is executed before an ON INT with low priority.

With the GOTO version there are also two effects, slightly differing —

- The branch is not taken until it has the highest priority of all pending branches.
- The execution of the branch does not lock out any other branches, so that at the end of the line to which it branches, if there are other pending branches, the highest one of those is executed.

For example, suppose there are these four statements in effect —

```
ON INT #4, 1 GOTO Routine_4
ON INT #5, 9 GOSUB Routine_5
ON INT #6, 5 GOTO 1000
ON INT #7, 15 GOSUB Routine_7
```

and also suppose that at the end of some BASIC line in the program, an interrupt had been received from all four of the interfaces involved. Then the process of dealing with them proceeds like this —

EVENT	NEXT ACTION	SYSTEM PRIORITY
Reaches end of current BASIC line	GOSUB Routine_7	Changes from 0 to 15
Finishes Routine_7	GOSUB Routine_5	Changes from 15 to 9

Suppose at this point another interrupt is received from select code 7.

EVENT	NEXT ACTION	SYSTEM PRIORITY
Reaches end of current BASIC line in Routine_5	GOSUB Routine_7	Changes from 9 to 15
Finishes Routine_7	Returns to interrupted point in Routine_5	Changes from 15 to 9
Finishes Routine_5	GOTO 1000	Changes from 9 to 0
Finishes with line 1000	GOTO Routine_4	Stays at 0

Environmental Considerations

Changes in program environment, i.e., calling a subprogram or returning from one, can affect whether an ON INT is in effect or not.

Once executed, the CALL version of an ON INT is **always** in effect, whether in the main program or in any subprogram, until it is redefined by another ON INT or is specifically disabled (see below).

In the GOSUB or GOTO versions, the statement is in effect **only** in the same program environment. This is to say that if you have executed an ON INT statement in your main program, then it is effective only while your program is executing part of the main program. The instant the program goes into a subprogram (through a CALL statement), the statement is no longer effective until the execution returns to the main program. Similarly, if you define an ON INT in a subprogram, it is effective only while the program is executing that subprogram.

A side-effect occurs here when you use the CALL version of an ON INT. By calling the subprogram with an ON INT, you have the effect of locking out the other interrupts, except those which are executed in the subprogram itself and other CALL versions. This is regardless of priority. In the priority example in the previous section, if the ON INT#5 had been a CALL instead of a GOSUB, then the second interrupt from select code 7 would not have been acknowledged until the subprogram had finished.

Since recursive calls of subprograms are possible, it is also possible that many calls to the same subprogram may be stacked up because an interrupt from a different select code with a CALL version of an ON INT in effect may be received while processing the CALL caused by a previous interrupt.

Disabling ON INT Branching

The branching enabled by an ON INT statement can be disabled using an OFF INT statement for the same select code. It is effective for the ON INT statement within the same program environment (main program or subprogram) or for the CALL versions of the ON INT within any environment.

The statement has the form —

```
OFF INT # {select code}
```

where {select code} is a numeric expression for any valid interface select code between 1 and 14, inclusive.

The effect of the OFF INT statement is to disable the ON INT for that select code within the current environment. If there is no ON INT statement currently in effect for the select code, then the OFF INT has no effect.

The DISABLE and ENABLE statements work the same way for the ON INT statements as they do for the ON KEY statements. They should not be confused with the DIR and EIR machine instructions, which disable and enable the interrupt system.

Mass Storage Activities

For devices meeting the operating system's criteria for mass storage peripherals, the reading and writing of records is simple.

If a device has been specified in a MASS STORAGE IS statement in BASIC, as in —

```
MASS STORAGE IS ":F"
```

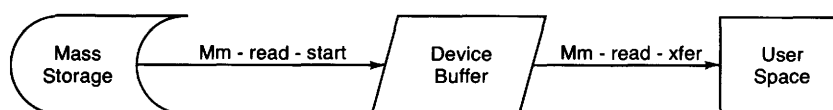
or is capable of being so specified, then it is possible to use utilities to access it.

There are two utilities involved in reading from a mass storage device — Mm_read_start and Mm_read_xfer — and there are two utilities involved in writing to a mass storage device — Mm_write_start and Mm_write_test. The reading utilities are always used together. So, too, are the writing utilities.

Reading from Mass Storage

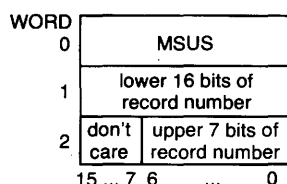
The flow of data to and from a mass storage device is buffered. For each device there is a “device buffer” in memory which holds data corresponding to a physical record (256 bytes). Device buffers are dynamically allocated by the operating system and their actual locations at any given time are of no concern.

To get information from a mass storage device into its device buffer takes the `Mm_read_start` utility. Then to get the information out of the buffer and into your user space takes the `Mm_read_xfer` utility. The transfer of data, therefore, looks something like this —

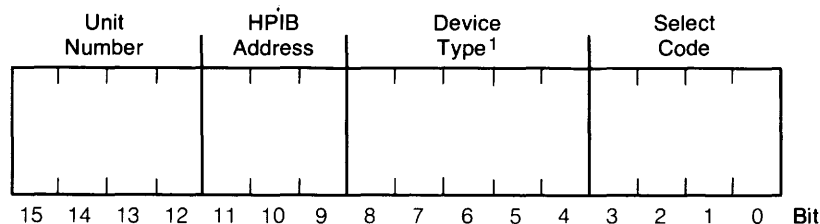


The utilities accomplish their purposes with the help of two locations containing vital information for their use. The first is the Mass Storage Descriptor (MSD) and the second is the Mass Storage Transfer Identifier (MSTID).

The MSD is three words in the ICOM region which contains the following information —



This information must be provided by your program. You must determine this information in advance of attempting the reading operation. The `msus` is of the form —



¹ The device type is the ASCII code for the type minus 1008.

The MSTID is a single word. The information in it is returned by the `Mm_read_start` utility and used by the `Mm_read_xfer` utility.

The usual procedure in reading a record from mass storage (which is all that can be read at one time) is to call the `Mm_read_start` utility and then, if all goes well with that, to call the `Mm_read_xfer` utility. Because the latter utility may have to wait on the operating system or the device, it is possible the utility may return without having completed the transfer. In that case, it is your option either to loop back and keep trying, or to do something else and try again later.

UTILITY: `Mm_read_start`

General Procedure: The record number is determined, then the transfer of the record's contents is made from the device to the device buffer. If the buffer allocation causes a memory overflow, there is an error.

Special Requirements: The record number and msus must be loaded into the MSD in advance of the call. There must be a stable location (not changed by other activities) for the MSTID to be held.

Calling Procedure:

1. Store the msus and record number into the MSD area.
2. Load register A with the address of the MSD area.
3. Call the utility.

Exit Conditions:

RET 1 Occurs if there is a memory overflow during execution of the utility.

RET 2 Occurs if all went normally. Register A contains the MSTID. This should be immediately stored in the location reserved for it.

UTILITY: Mm_read_xfer

General Procedure: The MSTID is used to retrieve the record from the device buffer. The record is stored into a location set aside for the purpose.

Special Requirements: The MSTID must be available from a previous call to Mm_read_start. A location of 128 consecutive words must be set aside to hold the contents of the record when they are returned by the utility.



Calling Procedure:

1. Load register A with the contents of the MSTID.
2. Load register B with the address of the storage location for the data.
3. Call the utility. The transfer may not be completed on the first or subsequent calls (see exit conditions). In that case, to successfully complete the transfer, all three steps must be repeated.

Exit Conditions:

- RET 1 Occurs when the transfer is not completed. It is up to your routine at this point to decide whether another attempt should be made immediately, or whether something else should be executed (and to come back later).
- RET 2 Occurs when the transfer is complete. The location specified contains the data. If register A contains a non-zero value, an error occurred and A is the error number. In addition to mass storage errors (80 through 99), error 19 is returned if the MSTID parameter is invalid.

CAUTION

Pressing RESET ( ) during execution of either of the above utilities may cause a SCRATCH A to occur.

The following is an example of a typical call to these utilities to read a record from mass storage —

```

ISOURCE Number:  BSS 2
ISOURCE Msd:     BSS 3
ISOURCE Mstid:   BSS 1
ISOURCE Record:  BSS 128
      .
      .
      .
ISOURCE      LDA #-1          ! Default MSUS
ISOURCE      STA Msd          ! Create the MSD
ISOURCE      LDA Number       ! Store low-order bits of record number
ISOURCE      STA Msd+1        ! Store high-order bits of record number
ISOURCE      LDA Number+1
ISOURCE      STA Msd+2
ISOURCE      LDA =Msd
ISOURCE      JSM Mm_read_start! From device to buffer
ISOURCE      JMP Memory_overflow
ISOURCE      STA Mstid         ! Keep the MSTID
ISOURCE Fetch:  LDA Mstid
ISOURCE      LDB =Record
ISOURCE      JSM Mm_read_xfer ! Transfer record to ICOM buffer
ISOURCE      JMP Fetch        ! Not completed (RET 1)
ISOURCE      SZA **2          ! Check for errors (RET 2)
ISOURCE      JSM Error_exit

```

Writing to Mass Storage

Writing to mass storage is very much like reading from it. The flow of data is buffered. To get the data from the user space into the device buffer, and then to transfer the data from the buffer to the mass storage device, the `Mm_write_start` utility is used. Then a test can be made to determine when the transfer is complete by using the `Mm_write_test` utility.

As with the reading utilities, these utilities accomplish their purposes with the help of the same two locations — MSD and MSTID. They contain the same information as they do in the reading utilities and are used in a similar fashion.

UTILITY: Mm_write_start

General Procedure: The record number is determined, then the transfer of the data is made from the ICOM region to the device buffer. If the buffer allocation causes a memory overflow, there is an error.

Special Requirements: The record number and msus must be loaded into the MSD in advance of the call. There must be a stable location (not changed by other activities) for the MSTID to be held. The data to be transferred must be ready (256 bytes — 128 consecutive words).

Calling Procedure:

1. Store the data to be transferred in its location. Store the msus and record number into the MSD area.
2. Load register A with the address of the MSD area.
3. Load register B with the address of the data location.
4. Call the utility.

Exit Conditions:

RET 1 Occurs if there is a memory overflow during execution of the utility.

RET 2 Occurs if all went normally. Register A contains the MSTID. This should be immediately stored in the location reserved for it.

UTILITY: Mm_write_test

General Procedure: The MSTID is used to check to see if the data from the buffer has been transferred to the mass storage device.

Special Requirements: The MSTID must be available from a previous call to Mm_write_start.



Calling Procedure:

1. Load register A with the contents of the MSTID.
2. Call the utility. The transfer may not be completed on the first or subsequent calls (see exit conditions). In that case, to successfully test for a completed transfer, both steps in the calling procedure must be repeated.

Exit Conditions:

- RET 1** Occurs when the transfer from the device buffer to the device is not completed. It is up to your routine at this point to decide whether another test should be made immediately, or whether something else should be executed (and to come back later).
- RET 2** Occurs when the transfer is complete. If register A contains a non-zero value, an error occurred and A is the error number. In addition to mass storage errors (80 through 99), error 19 is returned if the MSTID parameter is invalid.

CAUTION

Pressing RESET ( ) during execution of either of the above utilities may cause a SCRATCH A to occur.

The following is an example of a typical call to these utilities to write a record to mass storage —

```

ISOURCE Number:  BSS 2
ISOURCE Msd:     BSS 3
ISOURCE Mstid:   BSS 1
ISOURCE Record:  BSS 128
      .
      .
      .
ISOURCE      LDA #-1          ! Default MSUS
ISOURCE      STA Msd          ! Create the MSD
ISOURCE      LDA Number      ! Store low-order bits of record number
ISOURCE      STA Msd+1       ! Store high-order bits of record number
ISOURCE      LDA Number+1
ISOURCE      STA Msd+2
ISOURCE      LDA =Msd
ISOURCE      LDB =Record
ISOURCE      JSM Mm_write_start ! Put record in buffer
ISOURCE      JMP Memory_overflow
ISOURCE      STA Mstid        ! Keep the MSTID
ISOURCE Test:   LDA Mstid
ISOURCE      JSM Mm_write_test ! Is transfer of data complete?
ISOURCE      JMP Test        ! Not completed
ISOURCE      SZA #+2         ! Check for errors
ISOURCE      JSM Error_exit

```

System File Information

As an ASSIGN statement is executed in BASIC, a file-descriptor is created for that assignment in the operating system's files table. The ASSIGN statement essentially has two parameters — the file number and the file name (including the BASIC language mass storage unit specifier).

The file number is, for all practical purposes, an offset into the files table. The file name and the BASIC language mass storage unit specifier are translated and the critical information associated with them comprise an entry in the files table (i.e., the “file descriptor”).

The file descriptor consists of 10 words containing the following information —

Word	Description
0	Lower 16 bits of the address of the first physical record in the file
1	Number of logical records in the file
2	Current physical record number (i.e., an offset from the file's beginning.
3	Current word in physical record
4	Size of a logical record (in words)
5	Mass storage unit specifier (msus)
6	Buffer address
7	Check read status (0 = off, 1 = on)
8	Highest 7 bits of the first physical record in the file
9	(Reserved by the operating system)

Note that words 0,5 and 7 contain the information necessary to create an MSD. You may access a file descriptor through two utilities — `Get_file_info` to obtain the information, and `Put_file_info` to change the information.

NOTE

A files table is created for each BASIC “environment” (i.e., main program and subprograms). When access is made through utilities to the files table, the table accessed is the one associated with the BASIC environment which called the assembly language program.

UTILITY: Get_file_info

General Procedure: The utility is given the file number and the location of a place to store the file descriptor. It retrieves the designated descriptor and stores it, provided the file has been assigned.

Special Requirements: There must be a ten-word area available for the utility to store the information from the descriptor.

Call Procedure:

1. Load register A with the address of the ten-word area where you desire the information to be stored.
2. Load register B with the file number (an integer from 1 to 10).
3. Call the utility.

Exit Conditions:

RET 1 Occurs if the file has not been assigned by a BASIC ASSIGN statement.

RET 2 Occurs if all went normally.

Here is an example of a routine which has a file number passed to it, and then gets the file descriptor —

```

"
"
"
ISOURCE File_descriptor: BSS 10
ISOURCE File:          BSS 1
"
"
"
ISOURCE SUB
ISOURCE Parameter:    FIL
ISOURCE Routine:      LDA =File
ISOURCE               LDB =Parameter
ISOURCE               JSM Get_value      ! Get file number
ISOURCE               LDA =File_descriptor
ISOURCE               LDB File
ISOURCE               JSM Get_file_info   ! Get file descriptor
ISOURCE               JMP No_file_error   ! File not assigned
"
"
"

```


UTILITY: Put_file_info

General Procedure: The utility is given the file number and the location of the area containing the new file descriptor information. It stores that information into the files table as indicated by the file number, provided that the file has been assigned.

Special Requirements: The new pointer information must be stored in the designated area before calling the utility. This information must be in the correct form and location or file difficulties may ensue. Most of the information is normally returned by the “Get_file_info” utility and only a couple of words are changed to change the pointer in the file (e.g., the current record and word numbers). Only words 2, 3, and 7 should be changed in the descriptor.

Calling Procedure:

1. Load register A with the address of the ten-word area where the information is stored.
2. Load register B with the file number (an integer from 1 to 10).
3. Call the utility.

Exit Conditions:

RET 1 Occurs if the file has not been assigned by a BASIC ASSIGN statement.

RET 2 Occurs if all went normally.

Here is an example where the next physical record in a file is specified —

```
File:          BSS 1    ! File number
File_descriptor: BSS 10  ! File information
.
.
.
ISZ File_descriptor+2  ! Increment record number
LDA #0
STA File_descriptor+3  ! Set word to 0
LDA =File_descriptor
LDB File
JSM Put_file_info
JMP No_file_error      ! File not assigned
```

Printing

Two utilities are provided to enable you to gain access to the standard system printer — `Printer_select` and `Print_string`.

`Printer_select` enables you to set the standard system printer to a select code of your choosing. `Print_string` enables you to print a string to the standard printer.

Utility: `Printer_select`

General Procedure: The utility is given the select code to be assigned as the standard system printer and the desired printing width. The utility makes the assignment and returns with the previous values of both the select code and printer width.

Special Requirements: The select code value must be in the range of 0 through 17 for the utility to work properly. Neither the previous nor the selected printer should be on HPIB device.

Calling Procedure:

1. Load register A with the desired select code.
2. Load register B with the desired printer width.
3. Call the utility.

Exit Conditions: There are no error exits from the utility, so it always returns to the instruction following the call. Register A contains the value of the previous select, and register B contains the value of the previous printer width.

The utility can feasibly be used just to interrogate the current value of the printer's select code. However, a second call to the utility is needed in such cases to assure that the select is not changed by the first call. So, for example —

```
ISOURCE LDA = 16
ISOURCE LDB = 80
ISOURCE JSM Printer_select
ISOURCE STA Select_code
ISOURCE STB Printer_width
ISOURCE JSM Printer_select
```

This results in an unchanged printer specification and the values for the select code and width being stored in the ICOM area for future use.

Because of the possibility that a RESET (**CONT** **STOP**), or similar interruption, may occur between the first and second calls to the utility, it is recommended that the first call have a definite valid value for the select code in A (as above). In that way, should there indeed be an interruption, a valid select code for the printer can be assured.

Utility: Print_string

General Procedure: The utility is given the address of a string, and it prints that string to the standard system printer.

Special Requirements: The string to be printed must be in standard string format (see “Data Structures” in Chapter 3). The string must be no longer than 506 characters.

Calling Procedure:

1. Load register A with the address of the string to be printed.
2. Call the utility.

Exit Conditions:

RET 1 If a memory overflow occurs during execution of the utility.

RET 2 If the **STOP** key is pressed during execution of the utility.

RET 3 If all goes normally.

For example —

```

ISOURCE String:      DAT 13,"ERROR IN CALL"
"
"
"
ISOURCE              LDA = String
ISOURCE              JSM Print_string
ISOURCE              JMP Overflowerror      ! Overflow condition
ISOURCE              JMP Stop_routine      ! Stop key pressed
ISOURCE              NOP                   ! Normal exit

```

CAUTION

Pressing RESET (**CONT** **STOP**) during execution of the
Print_string utility may cause a SCRATCH A to occur.

Chapter 8

Debugging

Summary: This chapter describes techniques for isolating and correcting logic problems in assembly programs. Included in the discussion are techniques for stepping through programs, getting dumps, patching, and using the keyboard.

The assembly system has provided you with a number of BASIC language tools to help you debug your assembly language programs during their development stages.

These tools are for run-time debugging, so your source code must have been assembled into object code and stored in the ICOM region before attempting to use any of the debugging features detailed in this chapter.

There are three classes into which these tools fall: stepping through programs, dumps, and value checking. There is also an additional capability provided for the correction of some errors — patching.

The BASIC statements available for debugging are —

```
IBREAK  
IBREAK ALL  
IBREAK DATA  
ICHANGE  
IDUMP  
INORMAL  
IPAUSE OFF  
IPAUSE ON
```

and the following BASIC functions are available —



```
DECIMAL  
IADR  
IMEM  
OCTAL
```

Stepping Through Programs


“Logic” difficulties are some of the hardest problems to solve in debugging programs. In batch environments, the usual solution is to print the contents of variables at critical points in the program or to print dumps. The capabilities for both of these methods are provided. However, advantage has been taken of the interactive, “hands-on” nature of the 9835A/B and a feature has been added which allows you to execute the assembly statements individually. This permits you to examine the flow of the program as it executes rather than having to decipher a dump or trying to print the contents of specific variables at what you guess is the critical point.

If you are desirous of looking only at particular points in the program, or at particular variables, there is also the ability to establish “break points” for these items, so that your debugging routines can be invoked only when certain conditions arise. You can also establish different routines for different break points, adding to the flexibility.

Individual Instruction Execution


Normally, all BASIC lines, including the ICALL statement, act as a **unit**. That is to say, whenever you press the  key, the line which is currently executing is allowed to finish before the program is actually interrupted. Thus, if you press  during execution of the line —



```
100 LET A=1+1
```

the line finishes and the variable A contains the value 2. Then the  takes effect. The same is true of a line containing an ICALL statement.

For example, if you press  during the execution of —

```
120 ICALL Sort(A(*))
```

then the assembly routine completes before the  is honored. This is not always desirable; especially not during debugging of the assembly routine. It does not allow you to look at the execution of the routine to help you determine what may be going wrong.

The same problem occurs with the  key. Pressing  causes an entire BASIC line to be executed. Thus, if you stepped through line 120 as above, the entire routine Sort would be executed, and you would not be able to observe its execution on an instruction-by-instruction basis.

To permit you to analyze the execution of assembly language routines, an executable BASIC statement has been provided —

```
IPAUSE ON
```

Now, should you have the sequence in your program —

```
110 IPAUSE ON
120 ICALL Sort(A(*))
```

then pressing **PAUSE** during the execution of line 120 would cause program execution to be interrupted after completion of whatever machine instruction is being executed at the time. Further, the assembly language source line associated with the following instruction is displayed according to certain rules.



If the source lines are still in memory when you press **PAUSE** (e.g., you just assembled the object code which you are running), then the source line is displayed. If the source is no longer in memory (e.g., the object code was obtained through an ILOAD), then the instruction displayed is the result of a “reverse assembly”. If there is an operand with an instruction which is reverse assembled, then the octal value of that operand is displayed (this is because the reverse assembly process has no way of knowing what symbols you might have used to assemble the instruction originally).

After pressing **PAUSE**, should you press **CONT**, execution resumes normally. It is not necessary for you to do anything (such as cleaning up the registers, etc.) for execution to resume as if you had never interrupted it.



After pressing **PAUSE**, you may want to observe the flow of execution of your assembly routine. This can be done by successively pressing the **STEP** key. Each time the key is pressed, another machine instruction is executed and the assembly source line associated with the next machine instruction is displayed. You may continue this way for as long as you like — until you press **CONT** to allow processing to proceed uninterrupted until the end of the routine.

Of course, the **STEP** key can be used to step through the BASIC program as you are used to doing. That feature is unchanged. It is possible, therefore, to “step into” the assembly language routine from the BASIC (i.e., you need only **STEP** into line 120 above) and not have to use the **PAUSE** key at all.

In summary, IPAUSE ON allows two unique features —

- The  key can be used to halt execution within an assembled routine.
- The  key can be used to execute individual assembly language instructions.

Some key things to remember in using the IPAUSE ON facility —

- This is an execution-time debugging tool. You must be executing your previously-assembled object code with an ICALL statement.
- If the source code is available for display, it will be displayed, otherwise the line is “reverse assembled”.
- Utilities are not stepped instruction-by-instruction, but rather as a unit.
- The  key performs in BASIC just as before.
- Keeping the  key depressed causes repeated execution of the stepping function, the same as in BASIC.

By way of example, suppose you had the following source code —

```

10 DIM A$(10)
20 ICOM 100
30 IPAUSE ON
40 IASSEMBLE Extract
50 Loop: LINPUT A$
60 PAUSE
70 ICALL Extract(A$)
80 PRINT "<"A$;">"
90 GOTO Loop
100 ISOURCE      NAM Extract      ! Extracts part of a
110 ISOURCE      ! string preceding comma
120 ISOURCE      EXT Get_value,Put_value
130 ISOURCE String: BSS 6
140 ISOURCE      SUB
150 ISOURCE Parameter:STR
160 ISOURCE Extract: LDA =String      ! Retrieve string
170 ISOURCE      LDB =Parameter
180 ISOURCE      JSM Get_value
190 ISOURCE      LDB String      ! Initialize counter
200 ISOURCE      LDA =String      ! Initialize stack pointer
210 ISOURCE      SAL 1
220 ISOURCE      ADA String
230 ISOURCE      ADA =1
240 ISOURCE      STA C
250 ISOURCE      CBU
260 ISOURCE Loop: WBC A      ! Retrieve next character
270 ISOURCE      CPA =54B      ! Is it a comma?
280 ISOURCE      JMP Yes

```



```

290 ISOURCE      DSZ B          ! Decrement. Done?
300 ISOURCE      JMP Loop
310 ISOURCE      RET 1          ! No commas, no extractee
320 ISOURCE Yes:  ADB =-1       ! Found comma, extract
330 ISOURCE      STB String     !   by changing length
340 ISOURCE      LDA =String    !   then extracting
350 ISOURCE      LDB =Parameter
360 ISOURCE      JMP Put_value
370 ISOURCE      END Extract

```

Then the following would be the display lines you would see as you executed this program using the **STEP** key —

```

10  DIM A#[10]
20  ICOM 100
30  IPAUSE ON
40  IASSEMBLE Extract
50 Loop: LINPUT A#
?
12345,6789
160 00060 002025 ISOURCE Extract: LDA =String    ! Retrieve string
170 00061 006025 ISOURCE      LDB =Parameter
180 00062 142025 ISOURCE      JSM Get_value
190 00063 007756 ISOURCE      LDB String         ! Initialize counter
200 00064 002021 ISOURCE      LDA =String        ! Initialize stack pointer
210 00065 170600 ISOURCE      SAL 1
220 00066 023753 ISOURCE      ADA String
230 00067 022021 ISOURCE      ADA =1
240 00070 030016 ISOURCE      STA C
250 00071 070530 ISOURCE      CBU
260 00072 074760 ISOURCE Loop:  WBC A            ! Retrieve next character
270 00073 012016 ISOURCE      CPA =54B          ! Is it a comma?
290 00075 054001 ISOURCE      DSZ B             ! Decrement. Done?
300 00076 067774 ISOURCE      JMP Loop
260 00072 074760 ISOURCE Loop:  WBC A            ! Retrieve next character
270 00073 012016 ISOURCE      CPA =54B          ! Is it a comma?
290 00075 054001 ISOURCE      DSZ B             ! Decrement. Done?
300 00076 067774 ISOURCE      JMP Loop
260 00072 074760 ISOURCE Loop:  WBC A            ! Retrieve next character
270 00073 012016 ISOURCE      CPA =54B          ! Is it a comma?
290 00075 054001 ISOURCE      DSZ B             ! Decrement. Done?
300 00076 067774 ISOURCE      JMP Loop
260 00072 074760 ISOURCE Loop:  WBC A            ! Retrieve next character
270 00073 012016 ISOURCE      CPA =54B          ! Is it a comma?
280 00074 066004 ISOURCE      JMP Yes
320 00100 026012 ISOURCE Yes:  ADB =-1         ! Found comma, extract
330 00101 037740 ISOURCE      STB String        !   by changing length
340 00102 002003 ISOURCE      LDA =String       !   then extracting
350 00103 006003 ISOURCE      LDB =Parameter
360 00104 166007 ISOURCE      JMP Put_value
80  PRINT "<";A#;">"
<12345>
90  GOTO Loop
50 Loop: LINPUT A#

```

Note that the address of the instruction, as well as the octal value of the instruction, is displayed along with the source line.

This stepping facility can also be used, quite effectively, with the IBREAK statement (discussed below).

Should the IPAUSE ON facility be no longer desired, it can be turned off with —

```
IPAUSE OFF
```

The two statements can appear repeatedly in a program, allowing the stepping facility to be used in testing some programs but skipping over already proven programs. For example, suppose you had two programs — Sorta and Sortn — but the first was already tested and the second was not. Then this sequence might appear in your program —

```
110 IPAUSE OFF
120 ICALL Sorta(A$(*))
130 IPAUSE ON
140 ICALL Sortn(A$(*))
```

Stepping through this sequence results in lines 110, 120, and 130 executing without interruption, but line 140's call to Sortn would be executed instruction-by-instruction.

Executing IPAUSE ON when the facility is already in effect causes no change. Similarly, executing IPAUSE OFF when the facility is already off causes no change.

Both IPAUSE ON and IPAUSE OFF can be executed from the keyboard.


Setting Break Points

It is possible to define points in an assembly language routine where the execution should pause should it ever reach that point. These are called “break points”. They can be used to pause execution — allowing you to utilize the stepping activity described above in IPAUSE ON or to investigate the contents of variables, etc. They can also be used to allow branching to some BASIC routine, giving you the power of BASIC in doing some of your debugging.

Simple Pausing

To simply pause at a break point, you need to execute the following statement in advance of reaching that point (either in the program or from the keyboard) —

```
IBREAK {address}
```

where {address} is the assembled location¹ for the break point desired. Following execution of this statement, anytime the program execution reaches this address, it pauses. You may do any keyboard operations necessary at this point, or you may start stepping the program, (if IPAUSE ON has been executed), or you may resume execution using the  key. The address must have been assembled before the IBREAK is executed.

If you were to execute —

```
IBREAK Hook, 4
```

then every time the fourth word past assembly label “Hook” is reached during execution, the program execution pauses. If you were to execute —

```
IBREAK Hook+4
```

then Hook is assumed to be a BASIC variable, and the result of the expression is assumed to be an absolute address using whatever the value of Hook is when the statement is executed.

You can also specify the number of occurrences of reaching a break point before pausing should come into effect. This is done by executing —

```
IBREAK {address} : {counter}
```

where {counter} is a numeric expression; any variables within {counter} are BASIC variables. A pause occurs when {address} has been reached {counter} number of times. {counter} is reset after each pause.

When a break point is reached and a pause is to be taken, the pause takes place **before** execution of the contents of that address.

Transfers

Instead of just pausing at a break point, it is possible to branch to a BASIC routine. The intent of this facility is to give you access to BASIC’s capabilities, particularly the printing and variable-testing facilities, during your debugging efforts.

¹ See “Buzzwords” in Chapter 1 for the definition of “assembled location”.

The branch can be any of the three standard forms of BASIC branching —

```
IBREAK {address} [ ; {counter} ] CALL {subprogram}
IBREAK {address} [ ; {counter} ] GOSUB {line identifier}
IBREAK {address} [ ; {counter} ] GOTO {line identifier}
```

When either CALL or GOSUB has been designated, execution of the assembly language routine is suspended when {address} is reached. Then the designated subprogram or subroutine is executed. When that subprogram or subroutine is completed, then execution of the assembly language routine resumes with {address}.

When GOTO is specified, an unconditional branch is taken when {address} is encountered and execution of the assembly language routine is terminated.

{counter} performs the same as in the simple pausing form.

In the GOSUB and GOTO forms, there is an “environmental” restriction. The {line identifier} must be in the same BASIC environment (i.e., main program or subprogram) as that in which the IBREAK statement is executed. More on this in “Environments” below.

Environments

The GOSUB and GOTO types of break points are related to the BASIC “environment” (i.e., main program or subprogram) in which they are executed. Whenever an IBREAK statement of either type is encountered, the resulting break point is effective only for the environment in which the statement is located. The CALL version of break points is in effect in all environments.

For example —

```
200 SUB Test
210 IBREAK Hook GOTO Check_hook
    .
    .
    .
```

the break point established for “Hook” is good only in the subprogram “Test”. Leaving Test causes the break point to be cleared.

Executing an IBREAK statement from the keyboard is effective only for the environment executing at the time the statement is made. For example, if the following program lines had been executed —

```
200 SUB Test
210 PAUSE
```

and while the pause caused by line 210 is still in effect —

```
IBREAK Hook GOTO Check_hook
```

is executed, then the break point established for “Hook” is good only in the subprogram “Test”. As with the above, leaving Test causes the break point to be cleared.

If no program is executing when an IBREAK is executed from the keyboard, then the main program is considered to be the environment for the break point. If the program is replaced, as with a GET or a LOAD, then the break point is cleared.

Data Locations

Break points can also be established for data locations. This is done with —

```
IBREAK DATA {address}
```

In this case, {address} is presumed to be a data location referenced by other instructions. Whenever it is referenced by execution of some instruction, the pause occurs.

If you were to say —

```
IBREAK DATA Renras.
```

then whenever “Renras” is referenced, such as in —

```
LDA Renras
```

a pause would occur for that instruction.

A counter can also be specified with this form of break point —

```
IBREAK DATA {address} ; {counter}
```

{counter} is of the same form, and operates in an identical fashion, to the counter of the non-DATA form of break point.

Because the XFR machine instruction may access a particular location twice when it is executed, the break point on a data location may not operate correctly if the instruction referencing it is an XFR. The way to avoid this incorrect operation of the break point is to set {counter} to 2. (The only time this problem occurs is when the destination area for the XFR overlaps the origination area.)

Symmetry suggests that you should also be able to branch to BASIC routines with the DATA form of break point just as you can with the non-DATA form. And so you can —

```
IBREAK DATA {address} [ ; {counter} ] CALL {subprogram}
IBREAK DATA {address} [ ; {counter} ] GOSUB {line identifier}
IBREAK DATA {address} [ ; {counter} ] GOTO {line identifier}
```

They operate in an identical fashion to transfers of the non-DATA type and are under the same “environmental” restrictions.

In order to determine whether an address is being referenced, each instruction is “interpreted” (that is, analyzed for its components). Resultantly, a program runs much slower while an IBREAK DATA statement is in effect.

In addition to the pausing capability, using IBREAK DATA also allows trapping on “protected memory” violations (see “Protected Memory” section of this chapter).

IBREAK Everywhere

You may have a total of eight (8) break points (regardless of type) in effect at a given time, except for one extreme case. It may be desirable to establish a break point at every location in the ICOM region. This can be accomplished with —

```
IBREAK ALL
```

This statement overrides all other IBREAK statements and causes a pause before execution of every instruction in the ICOM region. There are also branching forms —

```
IBREAK ALL CALL {subprogram}
IBREAK ALL GOSUB {line identifier}
IBREAK ALL GOTO {line identifier}
```

Note, however, that there is no {counter} in any of these forms.

Number of Break Points

As was mentioned above, there can be no more than eight (8) IBREAK statements in effect at one time, that is to say within the same environment. And only one IBREAK ALL can be in effect at a given time.

In addition, there can only be one IBREAK or IBREAK DATA each in effect for a given {address}. Executing an IBREAK or IBREAK DATA with the same {address} as specified in an already effective IBREAK or IBREAK DATA statement causes the newly-executed statement to override the previous one. While there may be an IBREAK and IBREAK DATA both for the same {address}, the capability is not a useful one.

Clearing Break Points

There are a number of ways that break points can be cleared. One way as has already been mentioned, is leaving the BASIC environment, which clears any GOSUB or GOTO type of break points. Another way is to reassemble the module containing the break points. A third way is to execute an INORMAL statement. This statement has the form —

```
INORMAL {address}
```

After execution of the statement, whatever form of break point is established for the address (except IBREAK ALL) is cleared.

If {address} is omitted in this statement —

```
INORMAL
```

then all break points are cleared. This is the only way to clear an IBREAK ALL which may be in effect.

Protected Memory

An assembly language program is allowed to access only certain portions of memory during the process of stepping with the **STEP** key or when an IBREAK DATA statement is in effect. Should you try to step through a program which makes an access outside of the allowed memory, then an error results (number 187). The same is true if an IBREAK DATA statement is in effect. “Access” means jumping to or writing into memory.

The allowed memory is —

- The ICOM region.
- BASIC’s “value” area (the region where BASIC variables are stored).
- BASIC’s common area (the region where BASIC common variables are stored).
- The processor registers
- The temporary values stored in the base page (pre-defined symbol “Base_page”).
- The utilities.

All other memory is considered “protected” memory.

Dumps

A common tool of debugging is the memory “dump”. This is a print-out (or display) of the contents of selected locations in the memory. A typical use is to dump areas of the ICOM containing data so that the actual contents at some point during execution can be compared with the expected contents. All of this is in the hope that the comparison yields differences which give a clue as to the source of the difficulties being encountered.

This tool is provided through the IDUMP statement which has the form —

```
IDUMP {location} [ ; {location} [ ; ... ] ]
```

This statement can be placed in a program to be executed (perhaps as the result of a branching IBREAK statement) or it can be executed from the keyboard (perhaps during a pause caused by stepping or IBREAK).

Any number of {location}s can be specified. They can take a number of forms. The simplest is —

```
{address}
```

Thus, IDUMP {address} prints the contents of {address} to the current system printer. The contents are printed in their octal representation.

{location} can specify a whole range of addresses by using the form —

```
{address} TO {address}
```

With this form, the IDUMP statement prints the contents of all addresses starting with the first and ending the last specified {address}. If the second address is numerically smaller than the first, then a “wrap-around” through the end of memory into the top of memory is taken. For example, if you execute —

```
IDUMP 177776 TO 1
```

then the contents of four addresses would be printed — those for 177776, 177777, 0, and 1, in that order. Again, the contents are printed in their octal (base-8) representation.

Addresses are always specified in their octal representation, or symbolically (such as “Hook” or “Loop”). This is the same as for an assembled location, which is what {address} happens to be.

The output of the IDUMP statement is always printed to the current system printer. It is in octal form, unless otherwise specified. This specification is accomplished by preceding {address} with {mode selection}, which is one of the following —

ASC for ASCII character representation
 BIN for binary representation (base-2)
 DEC for decimal representation (base-10)
 HEX for hexadecimal representation (base-16)
 OCT for octal representation (base-8)

Thus, the general form of {location} is —

[{mode selection}] {address} [TO {address}]

As an example of all this, take the example program at the beginning of the chapter. If a couple of statements are added so that the main BASIC program reads —

```

10  DIM A$(10)
20  ICOM 100
30  IASSEMBLE Extract
40  IBREAK Loop GOSUB Dump
50  IDUMP 41 TO 104      ! Dump of ICOM region
60  PRINT
70 Loop:  INPUT A$
80  ICALL Extract(A$)
90  PRINT "<;A$;>"
100 GOTO Loop
110 !
120 ! Dump A,B registers in octal form,
130 ! string length in decimal form, and
140 ! and the string in character form
150 !
160 Dump:  IDUMP A TO B;DEC String;ASC String,1 TO String,8
170 PRINT
180 RETURN

```

then running it results in the following print-out —

```
000041: 000005 030462 031464 032454 033067 034071 022265 100003 022607 000012
000053: 021335 000001 100207 000000 000205 002025 006025 142025 007756 002021
000065: 170600 023753 022021 030016 070530 141714 012016 066004 054001 067774
000077: 170201 026012 037740 002003 006003 166007

000000: 000115 000012
000041: +00010
000042: 12345,6789#5%

000000: 000071 000011
000041: +00010
000042: 12345,6789#5%

000000: 000070 000010
000041: +00010
000042: 12345,6789#5%

000000: 000067 000007
000041: +00010
000042: 12345,6789#5%

000000: 000066 000006
000041: +00010
000042: 12345,6789#5%

<12345>
```

Value Checking

Value checking is a method of tracing the value of variables in your assembly language program using the interactive capabilities of the 9835A/B. You already have been introduced to break points and dumps in earlier sections. The capability of value checking serves as a useful adjunct to these procedures.

The value checking of assembly “variables” is similar to the monitoring of variables in BASIC during a debugging phase. Just as you would use a live-keyboard operation or judiciously placed PRINT statements to trace the execution of a program or the change in value of a variable in a BASIC program, so too can you use the monitoring tools for assembly programs.

Functions

Four additional functions are provided as extensions to BASIC which can be useful in the monitoring of values in an assembly language program. The four are —

```
DECIMAL
IADR
IMEM
OCTAL
```

They can be used as other than monitoring tools, but their descriptions here are primarily in that context. As functions, these items can be easily adapted for use in the special function keys.

DECIMAL

This function has the form —

```
DECIMAL ( {octal value} )
```

The function converts an octal integer value into its decimal representation. If the argument given is not octal, then an error (number 184) results.

This can be used as a quick, simple way of converting octal numbers into the more familiar decimal value. Being a function, it can be used anywhere any other BASIC numeric function can be used. Often you will find it useful in PRINT statements which are a part of subroutines called by break points.

OCTAL

This function is the converse of the DECIMAL function. Its role is to convert decimal values into their octal (base-8) representation. The function has the form —

```
OCTAL ( {decimal value} )
```

This can be used as a quick, convenient method of converting decimal numbers into their frequently used octal representations (a form which is useful because of its ready conversion into binary representation, and vice-versa).

The values resulting from this function must be treated with care. Though the result of the function is an octal representation, the value is still base-10. This difference is unimportant unless you are going to do arithmetic with the value resulting from the function.

As an example of this, suppose the decimal value 15 is to be converted into octal. The method is —

```
OCTAL (15)
```

and the resultant value is 17, the octal representation of 15. Now, if the result has 1 added to it, as with the expression —

```
OCTAL (15)+1
```

the ultimate result is 18. This can be a surprise since the usual octal arithmetic suggests that the result of $17_8 + 1$ be 20_8 . To get the proper octal result, the procedure is —

```
OCTAL (15+1)
```

Note also that the expression —

```
OCTAL (OCTAL (15)+1)
```

still does not yield 20.

IADR

This function yields the numeric value in octal representation of an assembled location. The form is —

```
IADR ( {assembled location} )
```

As an example, take the case of the example program at the beginning of this chapter. The result of —

```
IADR (Loop, 4)
```

is 76.

This function can be viewed as a convenient method of determining the address of a symbol, or of an offset from a symbol.

IMEM

This function is a quick, convenient way to look at the contents of a specific location in memory. The result is a numeric value, in octal representation, for the contents of a specified address. The form is —



```
IMEM ( {assembled location} )
```

The function is similar in many respects to the IDUMP statement. It is easiest, perhaps, to list the differences —

- IMEM is a function, where IDUMP is a statement.
- IMEM deals only with a single address, where IDUMP can deal with many.
- IMEM represents the value only in octal, where IDUMP can use many different representations.
- IMEM can be displayed and stored, where IDUMP can only be printed.

An obvious use for the function is in a routine called by an IBREAK statement. By using the function in such a manner, perhaps in a PRINT statement, you can ease the burden of checking variables from the keyboard. You can even use the value returned as a comparison against some set of limits so that you print only when the value exceeds those limits. There are many other possibilities for use.

Interrogating Registers and Flags

Interrogating the processor register A, B, P, R, Pa, Cb, Db, Dmapa, Dmama, Dmac, C, D, Ar2, SE, and Ar1 yields meaningful results only when execution of an assembly language subprogram has been suspended due to detection of a break point, or due to the use of the  or  keys (see Stepping Through Programs).

Further, the values of certain processor flags are stored in specific memory locations when a subprogram is suspended as described above. The flags are then available for interrogation as follows:

Decimal Carry	least significant bit of location 30 ₈
Overflow	least significant bit of location 31 ₈
Extend	most significant bit of location 31 ₈

It is important to note that interrogating an I/O register (R4, R5, R6, or R7) causes an input I/O bus cycle, using the current Pa register contents as the interface address. See Chapter 7 for details on the effects of such an action.

Patching

Patching is the practice of changing the contents of memory locations without re-assembling.

Patching as a standard procedure does not come highly recommended in the programming world. Nonetheless, there are circumstances which arise that occasionally suggest patching as the most profitable course of action.

To change a particular location in memory in the 9835A/B is not difficult. The statement to use is —

```
ICHANGE {assembled location} TO {octal expression}
```

After execution of the statement, the specified {assembled location} contains the specified octal value.

Changing the contents of a register is a common use of this facility. However, it should be remembered that attempting to change the contents of the I/O registers (R4, R5, R6, or R7) causes an output I/O bus cycle to occur, using the Pa register for the interface address. See Chapter 7 for details on the effects of such an action.

Chapter 9

Table of Contents

Errors and Error Processing

Types of Errors	189
Syntax-Time and Assembly-Time Errors	189
Run-Time Errors	190
Utility: Error_exit	191
Run-Time Messages	193
Assembly-Time Messages	195

Chapter 9

Errors and Error Processing

Summary: This chapter contains a discussion of Assembly Language ROM and other related errors, and what causes them. Included are methods for trapping errors and possible methods for correcting them.

Whether you are writing or accessing an assembly language routine, it is possible to encounter an error resulting from your actions. The intent of this chapter is to give some guidance as to how certain errors can be handled. It is not intended as a definitive checklist of what can go wrong, nor is it an exhaustive treatment of the means to correct the difficulties which are listed. Rather, it is meant as a reference for some of the things which can go wrong, what might cause them, and how to deal with them. Each programmer has a unique method of approaching the problem of error processing and there is no way to anticipate all of them. Even so, the following should offer some assistance in identifying the source of an error.

Not every machine error is covered here — only those directly related to writing or accessing assembly language routines. A complete listing of error messages (though not in the same detail as in this chapter) can be found in Appendix J.

Types of Errors

There are three types of errors associated with assembly language routines: those which occur during the writing (or entering) of the source code (called “syntax-time” errors); those which occur while assembling the source code (called “assembly-time” errors); and those which occur during the execution of an assembly language routine (called “run-time” errors). Some of these errors can be anticipated and trapped, others cannot.

Syntax-Time and Assembly-Time Errors

Syntax errors are caught when entering source code, usually with the message —

```
IMPROPER ISOURCE STATEMENT
```

The error can then be immediately corrected and the statement reentered. A side-effect of this entry-time check of the syntax is that the time required for assembly is greatly shortened over what it would be if syntax-checking were deferred until assembly.

Errors encountered during the assembly process are indicated by the assembler in three ways:

- The message —

```
ERROR 192 IN LINE nn
```

is displayed. **nn** is the line number of the IASSEMBLE statement. This is a fatal BASIC error, unless otherwise trapped.

- Each line in the source code containing an assembly error is printed on the current system printer. Included is the message —

```
**ERROR**
```

followed by the error type.

- The message —

```
ERRORS IN ASSEMBLY
```

follows the listing of the individual errors. The total number of errors is also printed.

An explanation of the individual assembly-time errors can be found at the end of this chapter.

Run-Time Errors

Run-time errors can sometimes be anticipated. They come at two distinct times, and your error processing is different depending upon which of those times are of concern. The times are “program development” and “production run”.

During program development, errors normally are handled using the debugging techniques detailed in Chapter 8. Care should be taken in recognizing errors during development. Not all of them are obvious or indicated by an error message — many simply lock up the machine.

During the running of production (debugged) routines, errors can be caused by the users of the routines. For instance, the user may inadvertently assign an argument a value of zero when that argument is to be used as a divisor within the assembly language routine. You should try to anticipate these usage errors and program procedures to trap them.

There are many alternatives for actions to take when your routine encounters and traps a usage error. For example, you may wish to assign a value to a particular return variable, or you may want to print a warning message, or, perhaps, to correct the value and proceed with the routine. Another method is to notify the user by issuing a BASIC error message. Such messages can be issued through the `Error_exit` utility discussed below.

Of course, you need to tell the users (in the documentation of the routine) what kind of errors can occur, when they can occur, and what to do about them.

UTILITY: `Error_exit`

The `Error_exit` utility provides you with the capability of aborting an assembly language routine by “creating” a BASIC error. Two types of BASIC errors can be created — “recoverable”, which can be trapped by a BASIC `ON ERROR` statement; and “non-recoverable” (or “fatal”), which cannot be trapped.

General Procedure: The utility is given the number of the error to be created. Then the utility is called with the `JSM` instruction, but no return is made to the original assembly language routine from the utility. Instead, the utility uses the information placed on the return stack to help create the error. The return stack is appropriately “cleaned up” and control is returned either to the BASIC driver (if the error is non-fatal) or to the operating system (if the error is fatal).

Special Requirements: Error numbers are passed to the utility in the A register. The value of the error number is placed in bits 0-14. Bit 15 is set if the error is to be non-recoverable. If bit 15 is not set, the error will be recoverable. Error numbers 32 762 through 32 767, with bit 15 set, are reserved by the operating system and should not be used.

Calling Procedure:

1. Load the error number into the A register.
2. Call the utility using the `JSM` instruction.

Exit Conditions: The utility returns control to the BASIC driver which called the routine, appropriately setting conditions so that ERRL, ERRM\$, and ERRN work as expected. Also triggers ON ERROR, if applicable.

The utility can be used anywhere in your assembly language, wherever you would like to abort the execution of the current assembly language routine and where you would like to indicate to BASIC what reason (error) caused the abortion.

For example, suppose somewhere in one of your assembly routines you wanted to abort the routine if a certain variable (Flag) is non-zero at a certain point. Suppose also that the variable, when non-zero, contained the error number, then your program could look like —

```
ISOURCE LDA Flag
ISOURCE SZA #2
ISOURCE JSM Error_exit
```

Similarly, there are some utilities which, when an error is encountered, return an error number in register A. In these case, a quick two-instruction sequence can give you an error-related abort. For example, the Rel_math utility is such a utility —

```
ISOURCE JSM Rel_math
ISOURCE SZA #2
ISOURCE JSM Error_exit
```

As an example of a fatal error, suppose the error desired is 8. The error sequence could be —

```
ISOURCE LDA #100010B
ISOURCE JSM Error_exit
```

Run-Time Messages

The following is a list of the system error messages you, or the users of your routines, may receive should something go wrong retrieving, using, or storing assembly language routines. A possible corrective action, or actions, is included in the discussion of the error.

- ERROR 1** ROM missing, or configuration error. To operate the 9835A/B, all system ROMs must be in place. In addition, to write assembly programs, the Assembly Execution and the Assembly Development ROMs must also be installed. Perform the system test if the problem persists.
- ERROR 2** Memory overflow. You may have specified an ICOM which is too large for your current available space. Some things to try: select a smaller ICOM size; execute SCRATCH C (if no important data remain in common), delete modules and reduce the ICOM size; segment your BASIC programs; segment your assembly programs. The error may also be caused by trying to load modules which are too large for the current ICOM region (either collectively or individually).
- ERROR 184** Improper argument in DECIMAL or OCTAL function. The OCTAL function has a range from -65535 to $+65535$. The DECIMAL function has a range for its arguments of -177777_8 to $+177777_8$.
- ERROR 185** Break Table overflow. A maximum of eight breaks can be established with the IBREAK statements and be in effect at one time. If eight breaks are in effect, then to allow other breaks to be established it is necessary to clear previous breaks using the INORMAL statement.
- ERROR 186** Undefined BASIC label or subprogram name used in IBREAK statement. When the IBREAK statement is executed, an undefined label or name is allowed, but when the break actually occurs, the label or name must exist.
- ERROR 187** Attempt to write into protected memory; or, an attempt to execute an instruction not in the ICOM region. This is the result of an attempt to branch outside of permissible areas or to change the contents of memory outside of the permissible areas. There is probably a difficulty in the logic of the program which needs to be corrected. This error only occurs when the **STEP** key is being used, an IBREAK DATA statement is in effect, or when using the ICHANGE function.

- ERROR 188 Label used in an assembled location not found. Symbolic addressing requires that all assembly symbols be resolved by execution time. This error probably results from a misspelling of a label or forgetting to assemble the module containing the label.
- ERROR 189 Doubly-defined entry point or routine. A module being assembled (with an IASSEMBLE statement) or loaded from mass storage (with an ILOAD statement) contains a SUB or ENT entry point with the same label as a SUB or ENT entry point within a module already resident within the ICOM region. Check the other routines for the duplicate occurrences.
- ERROR 190 Missing ICOM statement. You must include an ICOM statement to create your ICOM region before assembling or loading modules. Program an ICOM statement of adequate size and re-run the program
- ERROR 191 Module not found. The module indicated in an ISTORE or IASSEMBLE statement is not currently resident in the ICOM region. Check the module names used in your ISTORE statement to find the one which is missing from memory.
- ERROR 192 Errors in assembly. At least one error was encountered while assembling one of the modules in your IASSEMBLE statement.
- ERROR 193 Attempt to move or delete module containing an active interrupt service routine. This is the result of trying to reduce the size of the ICOM region (or to eliminate it), or trying to delete a module, when one of the affected modules contains an active interrupt service routine (ISR). The only ways to allow the action to take place are to SCRATCH A (which affects a number of other things) or to inactivate the ISR. To inactivate the ISR, consult the routine's documentation, or press Reset ().
- ERROR 194 IDUMP specification too large. The resulting dump would be more than 32 768 elements.
- ERROR 195 Routine specified in ICALL not found. You are specifying the wrong routine name or you are failing to load the correct module. Double check the documentation indicating the location and name of the routine.
- ERROR 196 Unsatisfied externals. Symbolic addressing requires that all references to symbols outside the current module be resolved at the time any routine within the current module is executed. This may possibly be a missing ENT instruction within another module.

- ERROR 197** Missing COM statement. The routine you are calling is expecting to find or place some of its data in common, but you are not providing the COM statement required. Add the appropriate COM statement in the BASIC program and re-run it.
- ERROR 198** BASIC'S common area does not correspond to assembly module requirements. The routine you have called is expecting to find or place some of its data in common, but your COM statement does not match up with the assembly COM declarations in either type or size. Check both the COM statement in the BASIC program and the COM declarations in the assembly routine.
- ERROR 199** Insufficient number of BASIC COM items. The routine you are calling is expecting to find or place some of its data in common, but your BASIC COM statement does not provide enough variables to satisfy the routine's needs. Check both the COM statement in the BASIC program and the COM declarations in the assembly routine.

Assembly-Time Messages

The following is a list of the assembler error messages you may receive while assembling a module. All of these errors cause a "fatal" error, which means that the assembly produced no object code. After the error has been corrected, it is necessary to re-assemble the module containing the error. A possible corrective action, or actions, is included in the discussion of the error.

- DD** Doubly-defined label. A label can only be defined once in a module. In addition, any label used in an EXT instruction is restricted from being used again as a label in the module. Check all spellings; change a label name to something else, if necessary.
- EN** END statement missing; or module name does not match. The END statement (in an ISOURCE statement) must be included to signify the end of a module. The name in the END statement must match the name used in the immediately preceding NAM statement. Particular ones to look out for: assembling more than one module at a time, but leaving out the END instruction between modules; or, the END instruction is after the BASIC program's END statement.

- EX Expression evaluation error. This is a result of a mismatch of element types in the operand of an instruction. The particular prohibited forms are: relocatable + relocatable; external \pm external; using the relocatable or external forms with the * or / operators. Check the spelling and type of your symbols in the expression.
- LT Literal pools full or out of range. You may have exhausted the storage given in your literal pool (LIT) declarations. In this case you should add more LIT declarations or increase the size of the ones you have. Another cause of the error can be using a literal in an instruction and there is no literal pool within 512 words of the instruction. Additionally, for some instructions, the assembler attempts to create an indirect reference automatically and requires a literal pool within 512 words of the instruction. In either case, add another literal pool (using a LIT instruction) within range.
- MO ICOM region memory overflow. The current module being assembled has caused object code generation which exceeds the current memory allowance for the ICOM region. Either you must re-run the current **main BASIC program** with a new ICOM statement increasing the ICOM size, or you must rearrange your assembly so that the module fits. This latter course can include deleting other modules or rewriting the abortive module so that it requires less memory.
- RN Operand out of range. Some instructions using indirection require a relocatable expression to evaluate to an address within 512 words of the current address. Skips must be no more than 32 words in either direction. The EXE instruction requires a register (0 to 31) and the instructions in the Stack Group require registers in the range of 0 to 7. Check to see that the operand used is within the range appropriate for the instruction. Also, check the spelling on all symbols to see that the right symbol was used.
- SQ Parameter declaration pseudo-instruction out of sequence. The ANY, FIL, INT, REL, SHO, and STR pseudo-instructions must follow a SUB or COM pseudo-instruction, or be a part of a group of such pseudo-instructions which follow a SUB or COM pseudo-instruction. Any other appearance of these can cause this error. It can also be caused if a SUB sequence does not terminate with a machine instruction with a label. Check to see that you have not inadvertently omitted the SUB or COM, or have placed another instruction in between the pseudo-instruction and its SUB or COM.

TP	<p>Incorrect type of operand used. Each instruction requires that its operand be of a certain type — relocatable or absolute. Check the type of all symbols used in the expression in the operand and see that they correspond to the type required by the instruction. If you are using a constant, check to see that a constant is allowed by the instruction.</p>
UN	<p>Undefined symbol. By the end of the assembly, all symbols must have been defined, either by use as a label on an instruction or as a symbol associated with a value through an EQU, EXT, or SET pseudo-instruction. A symbol not so defined, except those pre-defined by the assembler, and used in the assembly, causes this error. Check the spelling of all undefined symbols to make sure that you did not intend something else. The symbol otherwise has to be defined, either by label or EQU, EXT, or SET.</p>

Appendices

Table of Contents

Appendix A: ASCII Character Set

ASCII Character Codes	204
-----------------------------	-----

Appendix B: Machine Instructions

Detailed List	207
Bit Patterns and Timings	221
Alphabetic List	221
Approximate Numerical List	221

Appendix C: Pseudo-Instructions

223

Appendix D: Assembly Language BASIC Language Extensions Formal Syntax

225

Appendix E: Pre-Defined Assembler Symbols

231

Appendix F: Utilities

233

Appendix G: Writing Utilities

235

Appendix H: I/O Sample Programs

Handshake String Output	237
Handshake String Input	239
Interrupt String Output	241
Interrupt String Input	244
DMA String Output	247
DMA String Input	250
HP-IB Output/ Input Drivers	253
Real-Time-Clock Example	257

Appendix I: Demonstration Cartridge

Using the tape	261
Typing Aids	261

Appendix J: Error Messages

265

Mass Storage ROM Errors	269
Plotter ROM Errors	269
Assembly Language ROM Errors	270
Assembly Time Errors	271

Appendix K: Maintenance


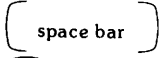
































































Maintenance Agreements	273
Sales & Service Offices	274

Appendix **A**

ASCII Character Set

The following table and chart show the ASCII character set and the keypresses necessary to obtain the ASCII character codes.

ASCII Character Set

ASCII Character	Comments	Key(s) to Press*	Octal Code	Decimal Code
NUL	Null	 	00	0
SOH	Start of Header	 	01	1
STX	Start of Text	 	02	2
ETX	End of Text	 	03	3
EOT	End of Transmission	 	04	4
ENQ	Enquiry	 	05	5
ACK	Acknowledgement	 	06	6
BEL	Bell	 	07	7
BS	Backspace	 	10	8
HT	Horizontal Tab	 	11	9
LF	Line Feed	 	12	10
VT	Vertical Tab	 	13	11
FF	Form Feed	 	14	12
CR	Carriage Return	 	15	13
SO	Shift Out	 	16	14
SI	Shift In	 	17	15
DLE	Data Link Escape	 	20	16
DC1	Device Control	 	21	17
DC2	Device Control	 	22	18
DC3	Device Control	 	23	19
DC4	Device Control	 	24	20
NAK	Negative Acknowledgement	 	25	21
SYN	Synchronous Idle	 	26	22
ETB	End of Text Block	 	27	23
CAN	Cancel	 	30	24
EM	End of Media	 	31	25
SUB	Substitute	 	32	26
ESC	Escape	 	33	27
FS	File Separator	  	34	28
GS	Group Separator	 	35	29
RS	Record Separator	 	36	30
US	Unit Separator	   *	37	31

* Assumes CAPS mode; multiple keys must be pressed simultaneously.

* Also can be found among calculator keys.

ASCII Character Set (continued)

ASCII Character	Comments	Key(s) to Press ☆	Octal Code	Decimal Code
SP	Blank	[space bar]	40	32
!	Exclamation Point	SHIFT [!]	41	33
"	Double Quote	SHIFT ["]	42	34
#	Pound Sign	SHIFT [#]	43	35
\$	Dollar Sign	SHIFT [\$]	44	36
%	Percent Sign	SHIFT [%]	45	37
&	Ampersand	SHIFT [&]	46	38
'	Apostrophe	[']	47	39
(Left Parenthesis	[(]	50	40
)	Right Parenthesis	[)]	51	41
*	Asterisk	SHIFT [*] *	52	42
+	Plus Sign	SHIFT [=] *	53	43
,	Comma	[<] *	54	44
-	Minus Sign (Dash)	[-] *	55	45
.	Period	[>] *	56	46
/	Forward Slash	[?] *	57	47
0	Numerics	[0] *	60	48
1		[1] *	61	49
2		[2] *	62	50
3		[3] *	63	51
4		[4] *	64	52
5		[5] *	65	53
6		[6] *	66	54
7		[7] *	67	55
8		[8] *	70	56
9		[9] *	71	57
:	Colon	SHIFT [:]	72	58
;	Semicolon	[;]	73	59
<	Less Than	SHIFT [<]	74	60
=	Equal	[=] *	75	61
>	Greater Than	SHIFT [>]	76	62
?	Question Mark	SHIFT [?]	77	63

☆ Assumes CAPS mode; multiple keys must be pressed simultaneously.

* Also can be found among calculator keys.

ASCII Character Set (continued)

ASCII Character	Comments	Key(s) to Press *	Octal Code	Decimal Code
@	Commercial At	SHIFT @ 2	100	64
A	Capital Letters	A	101	65
B		B	102	66
C		C	103	67
D		D	104	68
E		E	105	69
F		F	106	70
G		G	107	71
H		H	110	72
I		I	111	73
J		J	112	74
K		K	113	75
L		L	114	76
M		M	115	77
N		N	116	78
O		O	117	79
P		P	120	80
Q		Q	121	81
R		R	122	82
S		S	123	83
T		T	124	84
U		U	125	85
V		V	126	86
W		W	127	87
X		X	130	88
Y		Y	131	89
Z		Z	132	90
[Left Bracket	SHIFT [9	133	91
\	Reverse Slash	Inaccessible from Keyboard	134	92
]	Right Bracket	SHIFT] 0	135	93
↑	Up Arrow	^	136	94
_	Underscore	SHIFT - 6	137	95

* Assumes CAPS mode; multiple keys must be pressed simultaneously.

* Also can be found among calculator keys.

ASCII Character Set (continued)

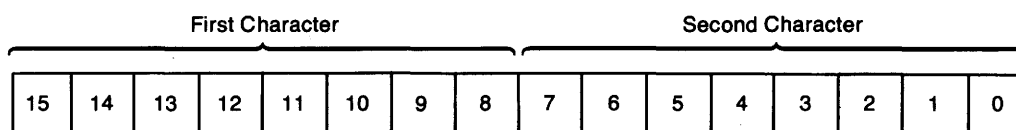
ASCII Character	Comments	Key(s) to Press*	Octal Code	Decimal Code
'	Grave Mark	Inaccessible from Keyboard	140	96
a	Noncapital Letters	SHIFT A	141	97
b		SHIFT B	142	98
c		SHIFT C	143	99
d		SHIFT D	144	100
e		SHIFT E	145	101
f		SHIFT F	146	102
g		SHIFT G	147	103
h		SHIFT H	150	104
i		SHIFT I	151	105
j		SHIFT J	152	106
k		SHIFT K	153	107
l		SHIFT L	154	108
m		SHIFT M	155	109
n		SHIFT N	156	110
o		SHIFT O	157	111
p		SHIFT P	160	112
q		SHIFT Q	161	113
r		SHIFT R	162	114
s		SHIFT S	163	115
t		SHIFT T	164	116
u		SHIFT U	165	117
v		SHIFT V	166	118
w		SHIFT W	167	119
x		SHIFT X	170	120
y		SHIFT Y	171	121
z		SHIFT Z	172	122
{	Left Brace	Inaccessible from Keyboard	173	123
	Vertical Line		174	124
}	Right Brace		175	125
~	Tilde		176	126
DEL	Delete		177	127

* Assumes CAPS mode; multiple keys must be pressed simultaneously.

ASCII Character Codes

ASCII Char.	EQUIVALENT FORMS			
	Binary	Oct	Hex	Dec
NULL	00000000	000	00	0
SOH	00000001	001	01	1
STX	00000010	002	02	2
ETX	00000011	003	03	3
EOT	00000100	004	04	4
ENQ	00000101	005	05	5
ACK	00000110	006	06	6
BELL	00000111	007	07	7
BS	00001000	010	08	8
HT	00001001	011	09	9
LF	00001010	012	0A	10
VT	00001011	013	0B	11
FF	00001100	014	0C	12
CR	00001101	015	0D	13
SO	00001110	016	0E	14
SI	00001111	017	0F	15
DLE	00010000	020	10	16
DC1	00010001	021	11	17
DC2	00010010	022	12	18
DC3	00010011	023	13	19
DC4	00010100	024	14	20
NAK	00010101	025	15	21
SYNC	00010110	026	16	22
ETB	00010111	027	17	23
CAN	00011000	030	18	24
EM	00011001	031	19	25
SUB	00011010	032	1A	26
ESC	00011011	033	1B	27
FS	00011100	034	1C	28
GS	00011101	035	1D	29
RS	00011110	036	1E	30
US	00011111	037	1F	31
space	00100000	040	20	32
!	00100001	041	21	33
"	00100010	042	22	34
#	00100011	043	23	35
\$	00100100	044	24	36
%	00100101	045	25	37
&	00100110	046	26	38
'	00100111	047	27	39
(00101000	050	28	40
)	00101001	051	29	41
*	00101010	052	2A	42
+	00101011	053	2B	43
,	00101100	054	2C	44
-	00101101	055	2D	45
.	00101110	056	2E	46
/	00101111	057	2F	47
0	00110000	060	30	48
1	00110001	061	31	49
2	00110010	062	32	50
3	00110011	063	33	51
4	00110100	064	34	52
5	00110101	065	35	53
6	00110110	066	36	54
7	00110111	067	37	55
8	00111000	070	38	56
9	00111001	071	39	57
:	00111010	072	3A	58
;	00111011	073	3B	59
<	00111100	074	3C	60
=	00111101	075	3D	61
>	00111110	076	3E	62
?	00111111	077	3F	63
@	01000000	100	40	64
A	01000001	101	41	65
B	01000010	102	42	66
C	01000011	103	43	67
D	01000100	104	44	68
E	01000101	105	45	69
F	01000110	106	46	70
G	01000111	107	47	71
H	01001000	110	48	72
I	01001001	111	49	73
J	01001010	112	4A	74
K	01001011	113	4B	75
L	01001100	114	4C	76
M	01001101	115	4D	77
N	01001110	116	4E	78
O	01001111	117	4F	79
P	01010000	120	50	80
Q	01010001	121	51	81
R	01010010	122	52	82
S	01010011	123	53	83
T	01010100	124	54	84
U	01010101	125	55	85
V	01010110	126	56	86
W	01010111	127	57	87
X	01011000	130	58	88
Y	01011001	131	59	89
Z	01011010	132	5A	90
[01011011	133	5B	91
\	01011100	134	5C	92
]	01011101	135	5D	93
^	01011110	136	5E	94
_	01011111	137	5F	95
`	01100000	140	60	96
a	01100001	141	61	97
b	01100010	142	62	98
c	01100011	143	63	99
d	01100100	144	64	100
e	01100101	145	65	101
f	01100110	146	66	102
g	01100111	147	67	103
h	01101000	150	68	104
i	01101001	151	69	105
j	01101010	152	6A	106
k	01101011	153	6B	107
l	01101100	154	6C	108
m	01101101	155	6D	109
n	01101110	156	6E	110
o	01101111	157	6F	111
p	01110000	160	70	112
q	01110001	161	71	113
r	01110010	162	72	114
s	01110011	163	73	115
t	01110100	164	74	116
u	01110101	165	75	117
v	01110110	166	76	118
w	01110111	167	77	119
x	01111000	170	78	120
y	01111001	171	79	121
z	01111010	172	7A	122
{	01111011	173	7B	123
	01111100	174	7C	124
}	01111101	175	7D	125
~	01111110	176	7E	126
DEL	01111111	177	7F	127

The following table gives the octal value for an ASCII character in the most significant byte (“First Character” column) and the least significant byte (“Second Character” column) of a word. The diagram illustrates the positions of the first and second character positions of a word.



ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent	ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
NUL	000000	000000	%	022400	000045
SOH	000400	000001	&	023000	000046
STX	001000	000002	'	023400	000047
ETX	001400	000003	(024000	000050
EOT	002000	000004)	024400	000051
ENQ	002400	000005	*	025000	000052
ACK	003000	000006	+	025400	000053
BEL	003400	000007	,	026000	000054
BS	004000	000010	-	026400	000055
HT	004400	000011	.	027000	000056
LF	005000	000012	/	027400	000057
VT	005400	000013	0	030000	000060
FF	006000	000014	1	030400	000061
CR	006400	000015	2	031000	000062
SO	007000	000016	3	031400	000063
SI	007400	000017	4	032000	000064
DLE	010000	000020	5	032400	000065
DC1	010400	000021	6	033000	000066
DC2	011000	000022	7	033400	000067
DC3	011400	000023	8	034000	000070
DC4	012000	000024	9	034400	000071
NAK	012400	000025	:	035000	000072
SYN	013000	000026	;	035400	000073
ETB	013400	000027	<	036000	000074
CAN	014000	000030	=	036400	000075
EM	014400	000031	>	037000	000076
SUB	015000	000032	?	037400	000077
ESC	015400	000033	@	040000	000100
FS	016000	000034	A	040400	000101
GS	016400	000035	B	041000	000102
RS	017000	000036	C	041400	000103
US	017400	000037	D	042000	000104
SP	020000	000040	E	042400	000105
!	020400	000041	F	043000	000106
"	021000	000042	G	043400	000107
#	021400	000043	H	044000	000110
\$	022000	000044	I	044400	000111

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
J	045000	000112
K	045400	000113
L	046000	000114
M	046400	000115
N	047000	000116
O	047400	000117
P	050000	000120
Q	050400	000121
R	051000	000122
S	051400	000123
T	052000	000124
U	052400	000125
V	053000	000126
W	053400	000127
X	054000	000130
Y	054400	000131
Z	055000	000132
[055400	000133
\	056000	000134
]	056400	000135
^	057000	000136
8	057400	000137
'	060000	000140
a	060400	000141
b	061000	000142
c	061400	000143
d	062000	000144

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
e	062400	000145
f	063000	000146
g	063400	000147
h	064000	000150
i	064400	000151
j	065000	000152
k	065400	000153
l	066000	000154
m	066400	000155
n	067000	000156
o	067400	000157
p	070000	000160
q	070400	000161
r	071000	000162
s	071400	000163
t	072000	000164
u	072400	000165
v	073000	000166
w	073400	000167
x	074000	000170
y	074400	000171
z	075000	000172
{	075400	000173
+	076000	000174
}	076400	000175
~	077000	000176
DEL	077400	000177

Appendix B

Machine Instructions

Detailed List

Instruction	Form	Group	Description	Page
AAR	AAR {n}	Shift/Rotate	Shifts the A register right the indicated number of bits with the sign bit filling all vacated bit positions. (Arithmetic right)	40
ABR	ABR {n}	Shift/Rotate	Shifts the B register right the indicated number of bits with the sign bit filling all vacated bit positions. (Arithmetic right)	40
ADA	ADA {loc} [, I]	Integer Math	Adds the contents of the specified location to the contents of register A. The result is in A. If a carry occurs, Extend is set, otherwise Extend is unchanged. If an overflow occurs, Overflow is set, otherwise Overflow is unchanged. A carry is from bit 15; an overflow is a carry from bit 15 or 14, but not both. Extend and Overflow are bits in the processor. Specifying register R4, R5, R6, or R7 as the location causes an input I/O bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.	35
ADB	ADB {loc} [, I]	Integer Math	Adds the contents of the specified location to the contents of register B. The result is in B. If a carry occurs, Extend is set, otherwise Extend is unchanged. If an overflow occurs, Overflow is set, otherwise Overflow is unchanged. A carry is from bit 15; an overflow is a carry from bit 15 or 14, but not both. Extend and Overflow are bits in the processor. Specifying register R4, R5, R6, or R7 as the location causes an input I/O bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.	35

Instruction	Form	Group	Description	Page
AND	AND {loc} [, I]	Logical	Logical “and” operation. The contents of the A register are compared, bit by bit, with the contents of the specified location. For each bit comparison a 1 results if both bits are 1’s, a 0 results otherwise. The 16-bit result is left in A. Specifying register R4, R5, R6, or R7 causes an input bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.	41
CBL	CBL	Stack	Clears the Cb register. Specifies the lower block of memory for byte-referencing stack instructions.	43
CBU	CBU	Stack	Sets the Cb register. Specifies the upper block of memory for byte-referencing stack instructions.	43
CDC	CDC	BCD Math	Clears Decimal Carry explicitly.	
CLA	CLA	Shift	Clears register A. This is exactly equivalent to SAR 16.	41
CLB	CLB	Shift	Clears register B. This is exactly equivalent to SBR 16.	41
CLR	CLR {n}	Load/Store	Clears the specified number of words, beginning at the location pointed at by the A register. A maximum of 16 words may be cleared.	34
CMA	CMA	Memory	Perform a one’s complement of the A register (bit by bit inversion of all 16 bits).	41
CMB	CMB	Memory	Perform a one’s complement of the B register (bit by bit inversion of all 16 bits).	41
CMX	CMX	BCD Math	Ten’s complement of Ar1. The mantissa of Ar1 is replaced with its ten’s complement and Decimal Carry is cleared.	45
CMY	CMY	BCD Math	Ten’s complement of Ar2. The mantissa of Ar2 is replaced with its ten’s complement and Decimal Carry is cleared.	46
CPA	CPA {loc} [, I]	Test/Branch	Compares the contents of register A with the contents of the specified location and skips if they are unequal. Indirect addressing may be specified. Specifying register R4, R5, R6, or R7 causes an input bus cycle to the interface addressed by the Pa register. {loc} must be on base or current page.	37

Instruction	Form	Group	Description	Page
CPB	CPB {loc} [, I]	Test/Branch	Compares the contents of register B with the contents of the specified location and skips if they are unequal. Indirect addressing may be specified. Specifying register R4, R5, R6, or R7 causes an input bus cycle to the interface addressed by the Pa register. {loc} must be on base or current page. {loc} must be on base or current page.	37
DBL	DBL	Stack	Clears the Db register. Specifies the lower block of memory for byte-referencing stack instructions.	43
DBU	DBU	Stack	Sets the Db register. Specifies the upper block of memory for byte-referencing stack instructions.	43
DDR	DDR	I/O	Disables Data Request. Cancels the DMA instruction.	47
DIR	DIR	I/O	Disables the interrupt system. Cancels the EIR instruction.	47
DMA	DMA	I/O	Enables the DMA mode. Cancels the DDR instruction.	47
DRS	DRS	BCD Math	Mantissa right shift of Ar1 for one digit. The twelfth digit is shifted into bits 0-3 of the A register. The non-digit part of the A register is cleared (bits 4-15), and the Decimal Carry bit in the processor is cleared. The first digit in the mantissa is set to 0.	45
DSZ	DSZ {loc} [, I]	Test/Alter/Branch	Decrements the contents of the specified location and skips if the new contents are 0. Specifying register R4, R5, R6, or R7 causes an input (or an input and an output) bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.	38
EIR	EIR	I/O	Enables the interrupt system. Cancels the DIR instruction.	47

Instruction	Form	Group	Description	Page
EXE	EXE {reg} [, I]	Miscellaneous	Executes the contents of a register. {reg} is an integer in the range of 0 through 31, indicating the register to be used (see Memory Map for the correspondence between location and register). The register is left unchanged unless the instruction code causes it to be altered. The next instruction to be executed is the one following the EXE, unless the code in the executed register causes a branch. Indirect addressing may be specified.	47
FDV	FDV	BCD Math	Fast divide. The mantissas of Ar1 and Ar2 are added together, along with Decimal Carry, until the first decimal overflow occurs. The result accumulates into Ar2. The number of additions without overflow is placed into the lower 4 bits of the B register (0-3). The remainder of the B register is cleared, as is the Decimal Carry bit in the processor.	46
FMP	FMP	BCD Math	Fast Multiply. Performs the multiplication by repeated additions. The mantissa of Ar1 is added to Ar2 along with Decimal carry, a specified number of times. The number of times is specified in the lower 4 bits (0-3) of the B register. The result accumulates in Ar2. If intermediate overflows occur, the number of times they occur appears in the lower 4 bits of the A register after the operation is complete. The upper 12 bits of the A register are cleared along with Decimal Carry.	46
FXA	FXA	BCD Math	Fixed-point addition. The mantissas of Ar1 and Ar2 are added together and the result placed in Ar2. Decimal Carry is used as the twelfth digit. After the addition, Decimal Carry is set if an overflow occurred, otherwise Decimal Carry is cleared.	46
IOR	IOR {loc} [, I]	Logical	Logical “inclusive or” operation. The contents of the A register are compared, bit by bit, with the contents of the specified location. For each bit comparison, a 0 results if both bits are 0's, a 1 otherwise. The 16-bit result is left in A. Specifying register R4, R5, R6, or R7 causes an input bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.	41

Instruction	Form	Group	Description	Page
ISZ	ISZ {loc} [, I]	Test/Alter/Branch	Increments the contents of the specified location and skips if the new contents are 0. Specifying register R4, R5, R6, or R7 causes an input (or an input followed by an output) bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.	38
JMP	JMP {loc} [, I]	Branch	Unconditionally branches to the specified location. Indirect addressing may be specified. {loc} must be on base or current page.	36
JSM	JSM {loc} [, I]	Branch	Jumps to subroutine. The value of the R register is incremented by 1 and the value of the P register (i.e., the location of the JSM instruction itself) is stored in the address pointed to by the R register. Execution then proceeds to the specified location. Return from the subroutine is effected by the RET instruction. Indirect addressing may be specified. {loc} must be on base or current page.	36
LDA	LDA {loc} [, I]	Load/Store	Loads register A with the contents of the specified location. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.	34
LDB	LDB {loc} [, I]	Load/Store	Loads register B with the contents of the specified location. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.	34
MLY	MLY	BCD Math	Mantissa left shift on Ar2 for one digit. This is a circular shift, with the bits 0-3 of the A register forming a thirteenth digit. The non-digit part of the A register is cleared (bits 4-15), and the Decimal Carry bit in the processor is cleared.	45

Instruction	Form	Group	Description	Page
MPY	MPY	Integer Math	Binary multiply. Uses Booth's Algorithm. The values of the A and B registers are multiplied together with the product placed into A and B. The A register contains the least significant bits and the B register contains the most significant bits and the sign. B may contain any integer value except - 32 768.	35
MRX	MRX	BCD Math	<p>Mantissa right shift on Ar1. The number of digits to be shifted is specified in the lower 4 bits (0-3) of the B register. The shift is accomplished in three stages:</p> <ol style="list-style-type: none"> 1) Bits 0-3 of the A register are right-shifted into D1 of the mantissa, with the twelfth digit being lost. This is the first shift. This shift always takes place, even if B = 0. 2) The digits are then right-shifted for the remaining number of digits specified. The twelfth digit is lost on each shift (except for the last shift) and the vacated digits are zero-filled. 3) Finally, the last right-shifting takes place, with the twelfth digit shifting into the lower 4 bits (0-3) of the A register. The Decimal Carry bit in the processor is cleared and the non-digit part of the A register is cleared (bits 4-15). 	44
MRY	MRY	BCD Math	<p>Mantissa right shift on Ar2. The number of digits to be shifted is specified in the lower 4 bits (0-3) of the B register. The shift is accomplished in three stages:</p> <ol style="list-style-type: none"> 1) Bits 0-3 of the A register are right-shifted into D1 of the mantissa, with the twelfth digit being lost. This is the first shift. This shift always takes place, even if B = 0. 2) The digits are right-shifted for the remaining number of digits specified. The twelfth digit is lost on each shift (except for the last shift) and the vacated digits are zero-filled. 	45

Instruction	Form	Group	Description	Page
			3) Finally, the last right-shifting takes place, with the twelfth digit shifting into the lower 4 bits (0-3) of the A register. The non-digit part of the A register is cleared (bits 4-15), and the Decimal Carry bit in the processor is cleared.	
MWA	MWA	BCD Math	Mantissa word addition. The contents of the B register are added to the ninth through twelfth digits of the Ar2 register. Decimal Carry is added to the twelfth digit; if an overflow occurs, Decimal Carry is set, otherwise Decimal Carry is cleared.	46
NOP	NOP	Miscellaneous	Null operation. This is exactly equivalent to LDA A.	47
NRM	NRM	BCD Math	Normalizes the Ar2 mantissa. Up to twelve left-shifts of the mantissa are performed until the first digit of the mantissa is non-zero. If the original first digit is already non-zero, no shifts occur. The number of shifts required is stored in the first 4 bits (0-3) of the B register. If 12 shifts are required, the Decimal Carry bit in the processor is set; otherwise, the Decimal Carry bit is cleared. The exponent is not altered.	45
PBC	PBC {reg} [, I]	Stack	Pushes the lower byte (right half) of the specified register onto the stack pointed at by the Cb and C registers. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing of the C register can be specified. Incrementing is the default. {reg} must be in the range of 0 through 7. The incrementing or decrementing action takes place before pushing.	43
PBD	PBD {reg} , D PBD {reg} [, I]	Stack	Pushes the lower byte (right half) of the specified register onto the stack pointed at by the Db and D registers. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing the D register can be specified. Incrementing is the default. {reg} must be in the range of 0 through 7. The incrementing or decrementing action takes place before pushing.	43

Instruction	Form	Group	Description	Page
PWC	PWC {reg} , D PWC {reg} [, I]	Stack	Pushes entire register (full word) onto the stack pointed at by the C register. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing the C register may be specified. Incrementing is the default. {reg} must be in the range of 0 through 7. The incrementing or decrementing action takes place before pushing.	43
PWD	PWD {reg} , D PWD {reg} [, I]	Stack	Pushes the entire register (full word) onto the stack pointed at by the D register. Specifying register R4, R5, R6, or R7 causes an input I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing the D register may be specified. Incrementing is the default. {reg} must be in the range of 0 through 7. The incrementing or decrementing action taken place before pushing.	43
RAL	RAL {n}	Shift/Rotate	Rotates the A register left the indicated number of bits. Bit 15 rotates into bit 0 (left circular). Maximum rotation of 16 bits.	40
RAR	RAR {n}	Shift/Rotate	Rotates the A register right the indicated number of bits. Bit 0 rotates into bit 15 (right circular). Maximum rotation of 16 bits.	40
RBL	RBL {n}	Shift/Rotate	Rotates the B register left the indicated number of bits. Bit 15 rotates into bit 0 (left circular). Maximum rotation of 16 bits rotated.	40
RBR	RBR {n}	Shift/Rotate	Rotates the B register right the indicated number of bits. Bit 0 rotates into bit 15 (right circular). Maximum rotation of 16 bits.	40
RET	RET {n}	Branch	Returns from subroutine. {n} is added to the contents of the address pointed to by the R register. The R register is decremented by 1. This is, in effect, a return from a JSM instruction (see above), to {n} instructions following the JSM itself. The "usual" return is RET 1. {n} must be in the range of - 32 through 31.	36

Instruction	Form	Group	Description	Page
RIA	RIA {adrs}	Test/Branch	Skips to {adrs} if register A is not 0, then increments register A by 1. Extend and Overflow are not effected by the incrementing action, even if a carry or overflow occurs. {adrs} must be within - 32 and + 31 of the current location.	37
RIB	RIB {adrs}	Test/Branch	Skips to {adrs} if register B is not 0, then increments register B by 1. Extend and Overflow are not affected by the incrementing action, even if a carry or overflow occurs. {adrs} must be within - 32 and + 31 of the current location.	37
RLA	RLA {adrs} [, S] RLA {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the least significant bit of the A register is not 0. Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	39
RLB	RLB {adrs} [, S] RLB {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the least significant bit of the B register is not 0. Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 the current location.	39
RZA	RZA {adrs}	Test/Branch	Skips to {adrs} if register A is not 0. {adrs} must be within - 32 and + 31 of the current location.	37
RZB	RZB {adrs}	Test/Branch	Skips to {adrs} if register B is not 0. {adrs} must be within - 32 and + 31 of the current location.	37
SAL	SAL {n}	Shift/Rotate	Shifts the A register left the indicated number of bits with all vacated bit positions becoming 0. Maximum shift is 16 bits.	40
SAM	SAM {adrs} [, S] SAM {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the A register is negative (bit 15 is 1). Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	38
SAP	SAP {adrs} [, S] SAP {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the A register is positive or zero (bit 15 is 0). Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	38
SAR	SAR {n}	Shift/Rotate	Shifts the A register right the indicated number of bits with all vacated bit positions becoming 0. Maximum shift is 16 bits.	40

Instruction	Form	Group	Description	Page
SBL	SBL {n}	Shift/Rotate	Shifts the B register left the indicated number of bits with all vacated bit positions becoming 0. Maximum shift is 16 bits.	40
SBM	SBM {adrs} [, S]	Test/Alter/Branch	Skips to {adrs} if the B register is negative (bit 15 is 1). Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	38
	SBM {adrs} [, C]	Test/Alter/Branch		
SBP	SBP {adrs} [, S]	Test/Alter/Branch	Skips to {adrs} if the B register is positive (bit 15 is 0). Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	38
	SBP {adrs} [, C]			
SBR	SBR {n}	Shift/Rotate	Shifts the B register right the indicated number of bits with all vacated bit positions becoming 0. Maximum shift is 16 bits.	40
SDC	SDC {adrs}	BCD Math	Skips to {adrs} if Decimal Carry is clear. Decimal carry is a single bit in the processor which may have been set by certain arithmetic operations. {adrs} must be within - 32 and + 31 of the current location.	46
SDI	SDI	I/O	Sets DMA inwards. Reads from peripheral, writes to memory.	47
SDO	SDO	I/O	Sets DMA outwards. Reads from memory, writes to peripheral.	47
SDS	SDS {adrs}	BCD Math	Skips to {adrs} if Decimal Carry is set. Decimal carry is a single bit in the processor which may have been set by certain arithmetic operations. {adrs} must be with - 32 and + 31 of the current location.	46
SEC	SEC {adrs} [, S]	Test/Alter/Branch	Skips to {adrs} if Extend is clear. Extend is a single bit in the processor which may have been set by certain arithmetic operations. Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	39
	SEC {adrs} [, C]			

Instruction	Form	Group	Description	Page
SES	SES {adrs} [, S] SES {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if Extend is set. Extend is a single bit in the processor which may have been set by certain arithmetic operations. Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	39
SFC	SFC {adrs}	I/O	Skips to {adrs} if the Flag line is false (clear). The Flag line is the one associated with a peripheral on the current select code (pointed to by the Pa register). {adrs} must be within - 32 and + 31 of the current location.	47
SFS	SFS {adrs}	I/O	Skips to {adrs} if the Flag line is true (set). The flag line is that associated with the peripheral on the current select code (pointed to by the Pa register). {adrs} must be within - 32 and + 31 of the current location.	47
SIA	SIA {adrs}	Test/Branch	Skips to {adrs} if register A is 0, then increments register A by 1. Extend and Overflow are not affected by the incrementing action, even if a carry or overflow occurs. {adrs} must be within - 32 and + 31 of the current location.	37
SIB	SIB {adrs}	Test/Branch	Skips to {adrs} if register B is 0, then increment register B by 1. Extend and Overflow are not affected by the incrementing action, even if a carry or overflow occurs. {adrs} must be within - 32 and + 31 of the current location.	37
SLA	SLA {adrs} [, S] SLA {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the least significant bit of the A register is 0. Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	38
SLB	SLB {adrs} [, C] SLB {adrs} [, S]	Test/Alter/Branch	Skips to {adrs} if the least significant bit of the B register is 0. Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	39

Instruction	Form	Group	Description	Page
SOC	SOC {adrs} [, S] SOC {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if Overflow is clear. Overflow is a single bit in the processor which may have been set by certain arithmetic operations. Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	39
SOS	SOS {adrs} [, S] SOS {adrs} [, C]	Test/Alter/Branch	Skips to {adrs} if the Overflow is set. Overflow is a single bit in the processor which may have been set by certain arithmetic operations. Setting or clearing the bit after the test can be specified. {adrs} must be within - 32 and + 31 of the current location.	39
SSC	SSC {adrs}	I/O	Skips to {adrs} if Status line is false (clear). The status line is the one associated with the peripheral on the current select code (pointed to by the Pa register). {adrs} must be within - 32 and + 31 of the current location.	47
SSS	SSS {adrs}	I/O	Skips to {adrs} if the Status line is true (set). The status line is the one associated with the peripheral on the current select code (pointed to by the Pa register). {adrs} must be within - 32 and + 31 of the current location.	47
STA	STA {loc} [, I]	Load/Store	Stores the contents of the A register into the specified location. Specifying register R4, R5, R6, or R7 causes an output bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.	34
STB	STB {loc} [, I]	Load/Store	Stores the contents of the B register into the specified location. Specifying register R4, R5, R6, or R7 causes an output bus cycle to the interface addressed by the Pa register. Indirect addressing may be specified. {loc} must be on base or current page.	34
SZA	SZA {adrs}	Test/Branch	Skips to {adrs} if register A is 0. {adrs} must be within - 32 and + 31 of the current location.	37
SZB	SZB {adrs}	Test/Branch	Skips to {adrs} if register B is 0. {adrs} must be within - 32 and + 31 of the current location.	37

Instruction	Form	Group	Description	Page
TCA	TCA	Integer Math	Performs a two's complement of the A register (one's complement, incremented by 1). If a carry occurs, Extend is set, otherwise Extend is unchanged. If an overflow occurs, Overflow is set, otherwise Overflow is unchanged. A carry is from bit 15; an overflow occurs when complementing -32768 . Extend and Overflow are bits in the processor.	35
TCB	TCB	Integer Math	Performs a two's complement of the B register (one's complement, incremented by 1). If a carry occurs, Extend is set, otherwise Extend is unchanged. If an overflow occurs, Overflow is set, otherwise Overflow is unchanged. A carry is from bit 15; an overflow occurs when complementing -32768 . Extend and Overflow are bits in the processor.	35
WBC	WBC {reg} [, D] WBC {reg} , I	Stack	Withdraws a byte from the stack pointed at by the Cb and C registers and places it into the lower byte (right half) of the specified register. Specifying register R4, R5, R6, or R7 causes an output I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing the C register can be specified. Decrementing is the default. {reg} must be in the range of 0 through 31. The incrementing or decrementing routine takes place after the withdrawal.	43
WBD	WBD {reg} [, D] WBD {reg} , I	Stack	Withdraws a byte from the stack pointed at by the Db and D registers and places it into the lower byte (right half) of the specified register. Specifying register R4, R5, R6, or R7 causes an output I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing the D register can be specified. Decrementing is the default. {reg} must be in the range of 0 through 31. The incrementing or decrementing routine takes place after the withdrawal.	43

Instruction	Form	Group	Description	Page
WWC	WWC {reg} [, D] WWC {reg} , I	Stack	Withdraws a full word from the stack pointed at by the C register and places it into the specified register. Specifying register R4, R5, R6, or R7 causes an output I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing of the C register can be specified. Decrementing is the default. {reg} must be in the range of 0 through 31. The incrementing or decrementing action takes place after the withdrawal.	43
WWD	WWD {reg} [, D] WWD {reg} , I	Stack	Withdraws a full word from the stack pointed at by the D register and places it into the specified register. Specifying register R4, R5, R6, or R7 causes an output I/O bus cycle to the interface addressed by the Pa register. Incrementing or decrementing of the D register can be specified. Decrementing is the default. {reg} must be in the range of 0 through 31. The incrementing or decrementing action takes place after the withdrawal.	43
XFR	XFR {n}	Load/Store	Transfers the specified number of words, from the location starting at the address pointed at by the A register to the location starting at the address pointed at by the B register. A maximum of 16 words can be transferred.	34

**Alphabetic List
Bit Patterns and Timings**

Instruction	Bit Pattern															Timing	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
AAR n	1	1	1	1	0	0	0	1	0	0	0	0	←	n-1		→	n+9
ABR n	1	1	1	1	1	0	0	1	0	0	0	0	←	n-1		→	n+9
ADA	D/I	0	1	0	0	B/C	← address →										19
ADB	D/I	0	1	0	1	B/C	← address →										19
AND	D/I	1	0	1	0	B/C	← address →										19
CBL	0	1	1	1	0	0	0	1	0	1	0	0	1	0	0	0	12
CBU	0	1	1	1	0	0	0	1	0	1	0	1	1	0	0	0	12
CDC	0	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0	11
CLA	1	1	1	1	0	0	0	1	0	1	0	0	1	1	1	1	25
CLB	1	1	1	1	1	0	0	1	0	1	0	0	1	1	1	1	25
CLR n	0	1	1	1	0	0	1	1	1	0	0	0	←	n-1		→	6n+16
CMA	1	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0	9
CMB	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	9
CMX	0	1	1	1	0	0	1	0	0	1	1	0	0	0	0	0	59
CMY	0	1	1	1	0	0	1	0	0	0	1	0	0	0	0	0	23
CPA	D/I	0	0	1	0	B/C	← address →										22
CPB	D/I	0	0	1	1	B/C	← address →										22
DBL	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	12
DBU	0	1	1	1	0	0	0	1	0	1	0	1	0	0	0	0	12
DDR	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	12
DIR	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	0	12
DMA	0	1	1	1	0	0	0	1	0	0	1	0	0	0	0	0	12
DRS	0	1	1	1	1	0	1	1	0	0	1	0	0	0	0	1	56
DSZ	D/I	1	0	1	1	B/C	← address →										25
EIR	0	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0	12
EXE	D/I	1	1	1	0	0	0	0	0	0	0	←	register			→	14
FDV	0	1	1	1	1	0	1	0	0	0	1	0	0	0	0	1	37+13B
FMP	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	42+13B (note 2)
FXA	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	40
IOR	D/I	1	1	0	0	B/C	← address →										19
ISZ	D/I	1	0	0	1	B/C	← address →										25
JMP	D/I	1	1	0	1	B/C	← address →										14
JSM	D/I	1	0	0	0	B/C	← address →										23
LDA	D/I	0	0	0	0	B/C	← address →										19
LDB	D/I	0	0	0	1	B/C	← address →										19
MLY	0	1	1	1	1	0	1	1	0	1	1	0	0	0	0	1	32
MPY	0	1	1	1	1	0	1	1	1	0	0	0	1	1	1	1	65+2T (note 3)
MRX	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	62+4B (note 4)
MRY	0	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0	33+4B (note 4)
MWA	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	28
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11
NRM	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0	23+Z (note 5)
PBC r	0	1	1	1	1	0	0	1	1/0	1	1	0	0	←	→		23
PBD r	0	1	1	1	1	0	0	1	1/0	1	1	0	1	←	→		23
PWC r	0	1	1	1	0	0	0	1	1/0	1	1	0	0	←	r	→	23
PWD r	0	1	1	1	0	0	0	1	1/0	1	1	0	1	←	→		23

Instruction	Bit Pat							
	15	14	13	12	11	10	9	8
RAL _n	1	1	1	1	0	0	0	1
RAR _n	1	1	1	1	0	0	0	1
RBL _n	1	1	1	1	1	0	0	1
RBR _n	1	1	1	1	1	0	0	1
RET	1	1	1	1	0	0	0	0
RIA	0	1	1	1	0	1	0	0
RIB	0	1	1	1	1	1	0	0
RLA	0	1	1	1	0	1	1	1
RLB	0	1	1	1	1	1	1	1
RZA	0	1	1	1	0	1	0	0
RZB	0	1	1	1	1	1	0	0
SAL _n	1	1	1	1	0	0	0	1
SAM	1	1	1	1	0	1	0	1
SAR _n	1	1	1	1	0	0	0	1
SBL _n	1	1	1	1	1	0	0	1
SBM	1	1	1	1	1	1	0	1
SBP	1	1	1	1	1	1	0	0
SBR _n	1	1	1	1	1	0	0	1
SDC	0	1	1	1	0	1	0	1
SDI	0	1	1	1	0	0	0	1
SDO	0	1	1	1	0	0	0	1
SDS	0	1	1	1	0	1	0	0
SEC	1	1	1	1	1	1	1	0
SES	1	1	1	1	1	1	1	1
SFC	0	1	1	1	0	1	0	1
SFS	0	1	1	1	0	1	0	0
SIA	0	1	1	1	0	1	0	1
SIB	0	1	1	1	1	1	0	1
SLA	0	1	1	1	0	1	1	0
SLB	0	1	1	1	1	1	1	0
SOC	1	1	1	1	0	1	1	0
SOS	1	1	1	1	0	1	1	1
SSC	0	1	1	1	1	1	0	1
SSS	0	1	1	1	1	1	0	0
STA	D/I	0	1	1	0	B/C	←	
STB	D/I	0	1	1	1	B/C	←	
SZA	0	1	1	1	0	1	0	1
SZB	0	1	1	1	1	1	0	1
TCA	1	1	1	1	0	0	0	0
TCB	1	1	1	1	1	0	0	0
WBC _r	0	1	1	1	1	0	0	1
WBD _r	0	1	1	1	1	0	0	1
WWC _r	0	1	1	1	0	0	0	1
WWD _r	0	1	1	1	0	0	0	1
XFR _n	0	1	1	1	0	0	1	1

Instruction	Bit Pattern															Timing													
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0													
RAL _n	1	1	1	1	0	0	0	1	1	1	0	0	← 15-n →				25-n												
RAR _n	1	1	1	1	0	0	0	1	1	1	0	0	← n-1 →				n+9												
RBL _n	1	1	1	1	1	0	0	1	1	1	0	0	← 15-n →				25-n												
RBR _n	1	1	1	1	1	0	0	1	1	1	0	0	← n-1 →				n+9												
RET	1	1	1	1	0	0	0	0	1	0	← skip →						16												
RIA	0	1	1	1	0	1	0	0	0	1							→						14						
RIB	0	1	1	1	1	1	0	0	0	1							→						14						
RLA	0	1	1	1	0	1	1	1	H/ \overline{H}	C/s							→						14						
RLB	0	1	1	1	1	1	1	1	H/ \overline{H}	C/s							→						14						
RZA	0	1	1	1	0	1	0	0	0	0							→						14						
RZB	0	1	1	1	1	1	0	0	0	0							→						14						
SAL _n	1	1	1	1	0	0	0	1	1	0							0	0	← n-1 →				n+9						
SAM	1	1	1	1	0	1	0	1	H/ \overline{H}	C/s	← skip →						14												
SAR _n	1	1	1	1	0	0	0	1	0	1	0	0	← n-1 →				n+9												
SBL _n	1	1	1	1	1	0	0	1	1	0	0	0	← n-1 →				n+9												
SBM	1	1	1	1	1	1	0	1	H/ \overline{H}	C/s	← skip →						14												
SBP	1	1	1	1	1	1	0	0	H/ \overline{H}	C/s							→						14						
SBR _n	1	1	1	1	1	0	0	1	0	1	0	0	← n-1 →				n+9												
SDC	0	1	1	1	0	1	0	1	1	1	← skip →						14												
SDI	0	1	1	1	0	0	0	1	0	0	0	0	1	0	0	0	12												
SDO	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	12												
SDS	0	1	1	1	0	1	0	0	1	1	← skip →						14												
SEC	1	1	1	1	1	1	1	0	H/ \overline{H}	C/s							→						14						
SES	1	1	1	1	1	1	1	1	H/ \overline{H}	C/s							→						14						
SFC	0	1	1	1	0	1	0	1	1	0							→						14						
SFS	0	1	1	1	0	1	0	0	1	0							→						14						
SIA	0	1	1	1	0	1	0	1	0	1							→						14						
SIB	0	1	1	1	1	1	0	1	0	1							← skip →						14						
SLA	0	1	1	1	0	1	1	0	H/ \overline{H}	C/s													→						14
SLB	0	1	1	1	1	1	1	0	H/ \overline{H}	C/s													→						14
SOC	1	1	1	1	0	1	1	0	H/ \overline{H}	C/s													→						14
SOS	1	1	1	1	0	1	1	1	H/ \overline{H}	C/s							→						14						
SSC	0	1	1	1	1	1	0	1	1	0	→						14												
SSS	0	1	1	1	1	1	0	0	1	0	→						14												
STA	D _i	0	1	1	0	B _C	← address →									19													
STB	D _i	0	1	1	1		B _C	← address →									19												
SZA	0	1	1	1	0	1		0	1	0	0	← skip →						14											
SZB	0	1	1	1	1	1	0	1	0	0	← skip →						14												
TCA	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	9												
TCB	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	9												
WBC _r	0	1	1	1	1	0	0	1	1/0	1	1	1	0	← r →			23												
WBD _r	0	1	1	1	1	0	0	1	1/0	1	1	1	1	← r →			23												
WWC _r	0	1	1	1	0	0	0	1	1/0	1	1	1	0	← r →			23												
WWD _r	0	1	1	1	0	0	0	1	1/0	1	1	1	1	← r →			23												
XFR _n	0	1	1	1	0	0	1	1	0	0	0	0	← n-1 →				12n+21												

Notes on bit patterns:

B/C (Base Page/Current Page)
C/S (Clear/Set)
D/I (Direct/Indirect)
H/ \overline{H} (Hold/Don't Hold)
I/D (Increment/Decrement)

All are coded 0/1 respectively

skip
address

} if the high bit in the field is 1, the
field is negative (2's complement)

Notes on timings:

All timings are maximum clock times. The clock rate is 6 megahertz. Clock times may vary up to ± 5% from the clock rate.

Any operation using register R4, R5, R6, or R7, should add 7 clock times.

Any operation using register R8, R9, R10, R11, R12, R13, R14, or R15 should add 5 clock times.

Maximum interrupt lockout time is 239.

Minimum interrupt lockout time is 2.

Maximum DMA lockout time is 10.

Minimum DMA lockout time is 2.

Interrupt execution is 36.

DMA read = 3 + 10n + lockout time
DMA write = 3 + 9n + lockout time

} n is the number of words
transferred during a request

Note 1. B is the current value in bits 0 through 3 of the B register.

Note 2. If bits 0 through 3 (B) of the B register are 0 then the total timing is 34.

Note 3. T is the total number of 0 → 1 and 1 → 0 transitions in the A register (using an imaginary 0 to the right of bit 0).

Note 4. B is the current value in bits 0 through 3 of the B register. If B = 0, then the total timing is 26.

Note 5. Z is the number of leading zeroes in the mantissa of Ar2. If Z = 12, then the total timing is 69.

**Approximate Numerical List
Bit Patterns**

Instruction	Bit Pattern															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LD ^{A/B}	D _I	0	0	0	A _B	Address Field										
CP ^{A/B}	D _I	0	0	1	A _B											
AD ^{A/B}	D _I	0	1	0	A _B											
ST ^{A/B}	D _I	0	1	1	A _B											
JSM	D _I	1	0	0	0											
AND	D _I	1	0	1	0											
1/0SZ	D _I	1	0	1/0	1											
IOR	D _I	1	1	0	0	Register Address										
JMP	D _I	1	1	0	1											
EXE	D _I	1	1	1	0	0	0	0	0	0	0	Register Address				
SD ^{O/I}	0	1	1	1	0	0	0	1	0	0	0					
E _D IR	0	1	1	1	0	0	0	1	0	0	0	1	E _D	0	0	0
DMA	0	1	1	1	0	0	0	1	0	0	1	0	0	0	0	0
DDK	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0
D _C B ^{U/L}	0	1	1	1	0	0	0	1	0	1	0	U _L	D _C	0	0	0
P _W W ^{C/D}	0	1	1	1	W _B	0	0	1	1/0	1	1	P _W	C _D	Register Address		
MWA	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
CM ^{V/X}	0	1	1	1	0	0	1	0	0	V _X	1	0	0	0	0	0
FXA	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0
XFR	0	1	1	1	0	0	1	1	0	0	0	0	N=# of words binary=(n-1)			
CLR	0	1	1	1	0	0	1	1	1	0	0	0				
NRM	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0
CDC	0	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0
FMP	0	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0
FDV	0	1	1	1	1	0	1	0	0	0	1	0	0	0	0	1
MRX	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0
DRS	0	1	1	1	1	0	1	1	0	0	1	0	0	0	0	1
MRY	0	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0
MLY	0	1	1	1	1	0	1	1	0	1	1	0	0	0	0	1
MPY	0	1	1	1	1	0	1	1	1	0	0	0	1	1	1	1
S ^{F/D} S _C	0	1	1	1	0	1	0	S _C	1	F _D	Skip Field if bit 5 is 0, then skip to (P+n), n=bits 0-4 if bit 5=1, then skip to (P-n), n=two's complement of bits 0-4					
R _S Z ^{A/B}	0	1	1	1	A _B	1	0	R _S	0	Z _I						
S _R L ^{A/B}	0	1	1	1	A _B	1	1	S _R	H _H	C _S						
SS ^{S/C}	0	1	1	1	1	1	0	S _C	1	0						
S ^{A/B} P _M	1	1	1	1	A _B	1	0	P _M	H _H	C _S	complemented skip field					
S ^{O/E} C _S	1	1	1	1	O _E	1	1	C _S	H _H	C _S						
RET	1	1	1	1	0	0	0	0	1	0	Shift Field in source, n=1-16 binary=(n-1) complemented shift					
TC ^{A/B}	1	1	1	1	A _B	0	0	0	0	0	1	0	0	0	0	0
CM ^{A/B}	1	1	1	1	A _B	0	0	0	0	1	1	0	0	0	0	0
CL ^{A/B}	1	1	1	1	A _B	0	0	1	0	1	0	0	1	1	1	1
A ^{A/B} R	1	1	1	1	A _B	0	0	1	0	0	0	0	Shift Field in source, n=1-16 binary=(n-1) complemented shift			
R _S A ^{A/B} R	1	1	1	1	A _B	0	0	1	R _S	1	0	0				
S ^{A/B} L	1	1	1	1	A _B	0	0	1	1	0	0	0	Shift Field in source, n=1-16 binary=(n-1) complemented shift			
R ^{A/B} L	1	1	1	1	A _B	0	0	1	1	1	0	0				

Appendix **C**

Pseudo-Instructions

The following table lists the available assembler pseudo-instructions with a short description of each, and the page number of the more detailed description listed elsewhere in this manual.

Instruction	Form	Description	Page
ANY	ANY	Specifies a common or subroutine declaration to be any type	112
BSS	BSS {expression}	Reserves a block of memory	56
COM	COM	Preface for assembly language common declarations	128
DAT	DAT {expression} [, {expression} [, ...]]	Defines data generators	57
END	END {name}	Designates the end of a module	17
ENT	ENT {symbol} [, {symbol} [, ...]]	Identifies entry points in the module	77
EQU	EQU {expression}	Defines a symbol	71
EXT	EXT {symbol} [, {symbol} [, ...]]	Identifies external entry points	77
FIL	FIL	Specifies a subroutine declaration to be a file number	110
HED	HED {comment}	Source listing control for top-of-page with change of heading	64
IFA	IFA	} Beginning of conditional assembly	66
IFB	IFB		
IFC	IFC		
IFD	IFD		
IFE	IFE		
IFF	IFF		
IFG	IFG		
IFH	IFH		
IFP	IFP {numeric expression}		
INT	INT [(*)]	Specifies a common or subroutine declaration to be an integer	110
LIT	LIT {expression}	Reserve memory for literals and links	74
LST	LST	Source listing control for enabling the listing	61
NAM	NAM {name}	Designates the beginning of a module	17
REL	REL [(*)]	Specifies a common or subroutine declaration to be full-precision	110
REP	REP {expression}	Repeats instructions	59
SHO	SHO [(*)]	Specifies a common or subroutine declaration to be short-precision	110
SKP	SKP	Source listing control for top-of-page	63
SPC	SPC {integer expression}	Source listing control for printing blank lines	65
STR	STR [(*)]	Specifies a common or subroutine declaration to be a string	110
SUB	SUB	Preface for a subroutine entry point	108
UNL	UNL	Source listing control for disabling the listing	61
XIF	XIF	End of a conditional-assembly block	66

Appendix **D**

Assembly Language BASIC Language Extensions Formal Syntax

The following is an alphabetical list of the BASIC Language extensions provided by the Assembly Language ROMs. For a full discussion of their semantical meanings and applications, consult the indicated pages in this manual.

Assembled Location (page 4)

```
{symbol} [ , {BASIC numeric expression} ]
{expression} [ , {BASIC numeric expression} ]
```

where:

{BASIC numeric expression} serves as a decimal offset from the given label or constant.

{symbol} is an assembly location. It may be either a label for a particular machine instruction (in which case the address of the associated instruction is used), or an assembler-defined symbol (in which case the associated absolute address is used), or a symbol defined by an EQU instruction (in which case the associated **value** is used).

{expression} may be a numeric expression or a string expression. If numeric, a decimal calculation is performed and the result is interpreted as an octal value; if the result is not an octal representation or an integer, an error results. If a string expression is used, the string must be interpretable as either an octal integer constant or a known assembly symbol (see {symbol} above).

DECIMAL Function (page 184)

```
DECIMAL ( {BASIC numeric expression} )
```


IADR Function (page 185)

```
IADR ( {assembled location} )
```

IASSEMBLE (pages 60-67)

```
IASSEMBLE {module} [ , {module} [ , ... ] ] [ ; {option} [ , {option} [ , ... ] ] ]
IASSEMBLE [ALL] [ ; {option} [ , {option} [ , ... ] ] ]
```

where {module} is the name of an existing module in the source program.

{option} may be any of the following:

```
A
B
C
D
E
EJECT
F
G
H
LINES {numeric expression}
LIST
P
XREF
```

IBREAK (pages 174-180)

```

IBREAK [ DATA ] {address} [ , {counter} ] [ CALL {subprogram} ]
IBREAK [ DATA ] {address} [ , {counter} ] [ GOSUB {line identifier} ]
IBREAK [ DATA ] {address} [ , {counter} ] [ GOTO {line identifier} ]
IBREAK ALL [ CALL {subprogram} ]
IBREAK ALL [ GOSUB {line identifier} ]
IBREAK ALL [ GOTO {line identifier} ]

```

where:

{address} is an assembled location.

{subprogram} is the name of a BASIC subprogram.

{counter} is a numeric expression.

{line identifier} is a line in the BASIC program.

ICALL (pages 107-111)

```

ICALL {routine} [ ( {data item} [ , {data item} [ , ... ] ) ]

```

where {routine} is the label associated with a SUB pseudo-instruction sequence and {data item} takes on the same forms and attributes as parameters in BASIC's CALL statement.

ICHANGE (page 187)

```

ICHANGE {assembled location} TO {octal expression}

```

ICOM (pages 19-22)

```

ICOM {integer constant}

```

IDELETE (pages 22-23)

```

IDELETE {module} [ , {module} [ , ... ] ]
IDELETE ALL

```

where {module} is the name of an existing module in the ICOM region.

IDUMP (pages 181-183)

```
IDUMP {location} [ ; {location} [ ; ... ] ]
```

where {location} has the following syntax:

```
[ {mode selection} ] {address} [ TO {address} ]
```

with {address} an assembled location and {mode selection} taking on any of the following forms —

ASC	for ASCII character representation
BIN	for binary representation
DEC	for decimal representation
HEX	for hexadecimal representation
OCT	for octal representation

ILOAD (page 22)

```
ILOAD {file specifier}
```

where {file specifier} is of the same form as elsewhere in BASIC (see Mass Storage Techniques manual, or Operating and Programming manual).

IMEM Function (page 186)

```
IMEM ( {assembled location} )
```

INORMAL (page 179)

```
INORMAL [ {address} ]
```

where {address} is an assembled location.

IPAUSE OFF (page 174)

```
IPAUSE OFF
```

IPAUSE ON (pages 171-174)

```
IPAUSE ON
```

ISOURCE (pages 49-54)

```
ISOURCE {source line}
```

where {source line} may take either of the following forms —

```
[ {label} : ] {action} [ ! {comment} ]
[ {label} : ] ! {comment}
```

and:

{label} is of the same form as elsewhere in BASIC;

{action} is a machine instruction, pseudo-instruction, or data generator;

{comment} is any combination of characters

ISTORE (pages 23-24)

```
ISTORE {module} [ , {module} [ , ... ] ] ; {file specifier}
ISTORE [ALL]; {file specifier}
```

where:

{module} is the name of a module currently existing in the ICOM region.

{file specifier} is of the same form as elsewhere in BASIC (see the Mass Storage Techniques manual or the Operating and Programming manual).

LITERALS (pages 72-75)

```
= {expression} [ , {expression} [ , ... ] ]
```

{expression} may be absolute or relocatable

OCTAL Function (page 184)

```
OCTAL ( {numeric expression} )
```

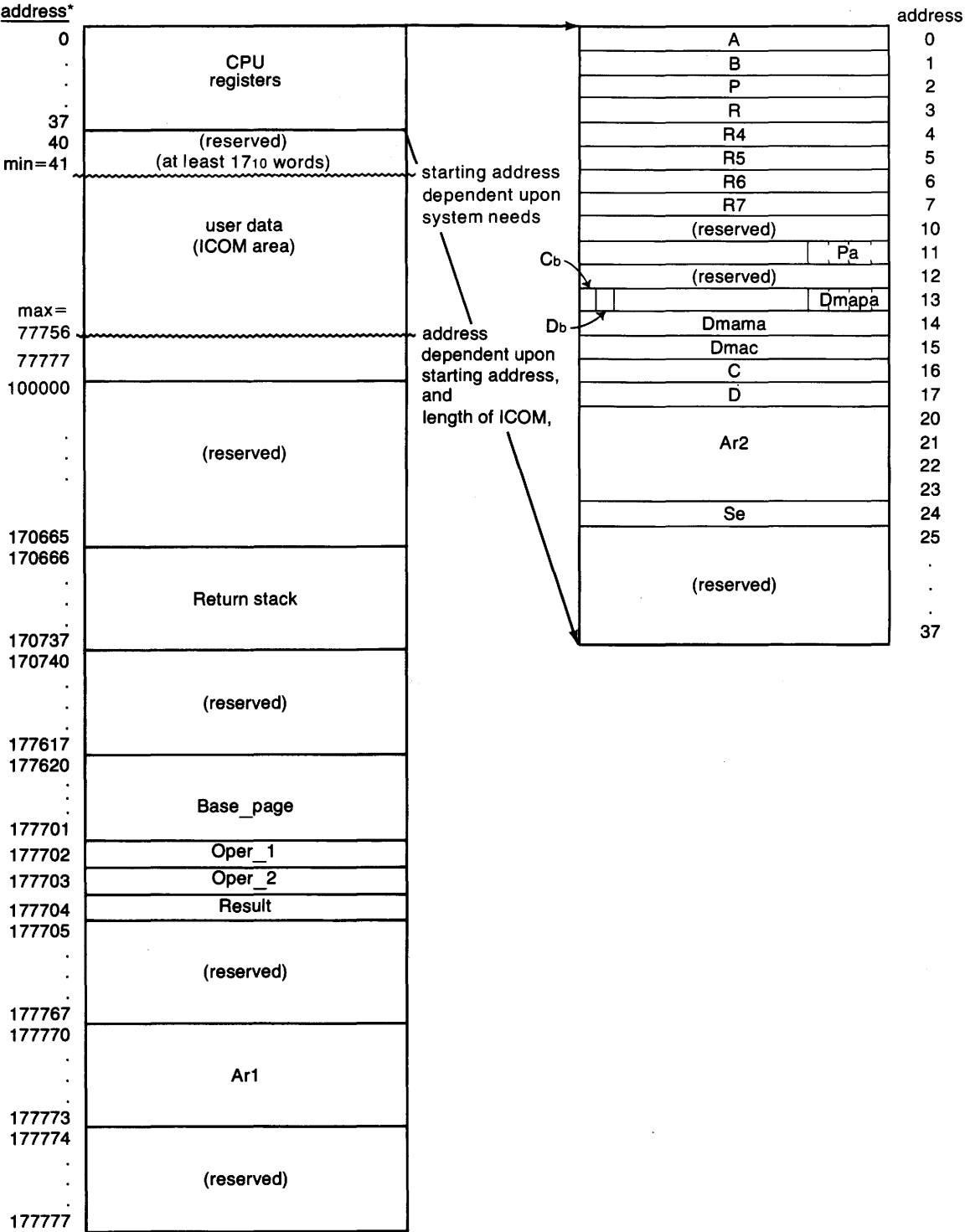

Appendix E

Predefined Assembler Symbols

The assembler has predefined a number of symbols and has reserved them as references to special locations in memory. Each of these locations has a special meaning and function. You may not redefine these symbols. They are —

Name	Description
A	Arithmetic accumulator
Ar1	} BCD arithmetic accumulators
Ar2	
B	Arithmetic accumulator
Base_page	Base page temporary area (50 words)
C	Stack pointer
Cb	Block bit for byte pointer in C (most significant bit of address 13s)
D	Stack pointer
Db	Block bit for byte pointer in D (second most significant bit of address 13s)
Dmac	DMA count register
Dmama	DMA memory address register
Dmapa	DMA peripheral register (lower 4 bits of address 13s)
End_isr_high	} Reserved symbols for use with interrupt service routines
End_isr_low	
Isr_flag	
Isr_psw	
Oper_1	} Arithmetic utility operand address registers
Oper_2	
P	Program counter
Pa	Peripheral address register (lower 4 bits of address 11s)
R	Return stack pointer
R4	} I/O registers
R5	
R6	
R7	
Result	Arithmetic utility result address register
Se	Shift-extend register
Utlcount	} Reserved symbols for writing utilities
Utlend	
Utltemps	

Each predefined symbol references a particular location in memory, except for the Utlend symbol, which refers to an execution address of a system routine. A graphical representation of these locations, plus others of interest, is presented on the next page.




* in octal representation

Utility Name	LDA with:	LDB with:	Exits	Other	Description	Page
Busy	address of bit pattern	address of parameter	RET 1		Retrieves busy bits for a BASIC variable	130
Error_exit	error number	N/A	None — returns to BASIC		Aborts execution of ICALL statement, setting an error number	191
Get_bytes	address of storage area	address of parameter	RET 1	Storage area consists of: 1st word — starting byte 2nd word — number of bytes to be transferred 3rd word on — sufficient space for string	Accesses substrings (or parts of arguments)	119
Get_elem_bytes	address of storage area	address of array info	RET 1	Array info obtained by Get_info utility. Relative element number must be stored in array pointer (word 16) of array info. Storage area same as in Get_bytes.	Same as “Get_bytes” used for accessing elements of string arrays	120
Get_file_info	address of storage area	file number	RET 2 — normal RET 1 — file unassigned	Storage area contents after return: word 0 — lower 16 bits of file address word 1 — number of defined records word 2 — current record number word 3 — current word in current record word 4 — size of defined record word 5 — mass storage unit specifier word 6 — buffer address word 7 — check read (0=off, 1=on) word 8 — high 7 bits of file address word 9 — (reserved by system)	Accesses a file-pointer	164
Get_info	address of storage area	address of array info	RET 1	Storage area must be at least: 3 words — simple variables 18 words — arrays for arrays, add 3 words for each 64K bytes in your machine's memory	Returns the characteristics of a variable passed as a parameter or existing in common	114
Get_element	address of storage area	address of parameter	RET 1	Array info obtained by Get_info utility. Relative element_number must be stored in array pointer (word 16) of array info. Storage area must be sufficient size to hold value.	Same as “Get_value”, used for elements in an array	118
Get_value	address of storage area	address of parameter	RET 1	Storage area must be sufficient size to hold value	Returns the value of a BASIC variable	117
Int_to_rel	N/A	N/A	RET 1	Load address of integer into Oper_1 and address of storage area into Result. Storage area must be at least 4 words.	Data type conversion from integer to full-precision	104
Isr_access	address of ISR	select code in bits 0-3; access code in bits 4-5; trial counter bits 8-14	RET 1 — linkage not established for reason found in register A: — 1 = resources unobtainable — 2 = select code linked to another ISR RET 2 — normal	select code is 0-7 for low-level or 8-15 for high-level; resource code is: 0 — no resources 1 — asynchronous access 2 — asynchronous access with DMA 3 — synchronous access trial counter is number of attempts before aborting (RET 1, with A set to — 1)	Establishes linkages for interrupts	143
Mm_read_start	address of mass storage descriptor	N/A	RET 1 — memory overflow RET 2 — normal (A contains mass storage transfer ID)	Mass storage descriptor is 3 words containing: word 1 — mass storage unit specifier word 2 — least significant 16 bits of record number word 3 — most significant 7 bits of record number	Prepares to read a physical record from mass storage	158

Appendix F

Utilities

Utility Name	LDA with:	LDB with:	Exits	Other	Description	Page
Mm_read_xfer	mass storage transfer ID	address of storage area	RET 1 — transfer incomplete RET 2 — transfer complete (A contains 0, or error number encountered during transfer)	Storage area must be at least 128 words Mass storage transfer ID would be returned from Mm_read_start utility. Storage area receives transferred information	Reads a physical record from mass storage	159
Mm_write_start	address of mass storage descriptor	address of storage area	RET 1 — memory overflow RET 2 — normal (A contains mass storage transfer ID)	Mass storage descriptor same as in Mm_read_start. Storage area must be at least 128 words and contain information to be transferred	Writes a physical record to mass storage	161
Mm_write_test	mass storage transfer ID	N/A	RET 1 — transfer incomplete RET 2 — transfer complete (A contains 0, or error number encountered during transfer)	Mass storage transfer ID is returned from Mm_write_start utility.	Verifies a physical record was written to mass storage	161
Printer_select	select code	printer width	RET 1 (A contains previous printer select code; B contains previous printer width)		Changes or interrogates select-code for standard printer	166
Print_string	address of string	N/A	RET 1 — memory overflow RET 2 —  pressed RET 3 — normal	String must be in same form as standard string	Outputs a string to the standard printer	167
Put_bytes	address of storage area	address of parameter	RET 1	Storage area same as Get_bytes	Replaces substrings (or parts of arguments)	124
Put_elem_bytes	address of storage area	address of array info	RET 1	Same as Get_elem_bytes	Same as "Put_bytes", used for accessing elements of string arrays	125
Put_element	address of storage area	address of array info	RET 1	Same as Get_element	Same as "Put_value", used for elements in an array	123
Put_file_info	address of storage area	file number	RET 1 — file unassigned RET 2 — normal	Same as Get_file_info	Manipulates a file-pointer	165
Put_value	address of storage area	address of parameter	RET 1		Changes the value of a BASIC variable	122
Rel_math	number of operands	execution address	RET 1 (A contains 0, or an error number)	Address of first operand into Oper_1 and address of second operand into Oper_2. Address of result area into Result. Execution address is for the desired routine.	Provides access to all the arithmetic routines	99
Rel_to_int	N/A	N/A	Overflow bit may be set	Address of the value to be converted should be stored in Oper_1, address of storage area of integer into Result	Data type conversion from full-precision to integer	102
Rel_to_sho	N/A	N/A		Address of the value to be converted should be stored in Oper_1; address of storage area for converted number should be stored in Result	Data type conversion from full-precision to short	103
Sho_to_rel	N/A	N/A		Same as Rel_to_sho	Data type conversion from short-precision to full	105

Appendix G

Writing Utilities

A utility, essentially, is a “special” assembly language subroutine. What makes it special is a set of instructions which keeps it from being displayed when a program is being stepped through using the **STEP** key. This provides some manner of security for the code in the routine from the casual user.

The following must be done to make a section of code into a utility —

1. The entry point for the utility must consist of the instruction —

```
ISZ Ut1count
```

2. Each exit point from the utility must consist of the following instructions —

```
DSZ Ut1count
```

```
RET n (n may be any number, - 32 through + 31, depending upon the desired  
returning point)
```

```
JSM Ut1end
```

For example, here is a simple utility to increment a private counter —

```
ISOURCE User_counter: BSS 1
      .
      .
      .
ISOURCE Users: ISZ Ut1count
ISOURCE      ISZ User_counter
ISOURCE      DSZ Ut1count
ISOURCE      RET 1
ISOURCE      JSM Ut1end
      .
      .
      .
```

It is not required that a utility actually be a subroutine. It may also be in-line code by replacing the RET with JMP *+2. By making a section of in-line code a utility, you can make your **STEP** actions in debugging simpler. If you already know what a section does and don't want to have to step through each instruction in that section each time it is encountered, you can make it into a utility as above. Then, whenever it is encountered, the section is stepped through as if it were a single statement.

Utilities, and calls to utilities, are not allowed in interrupt service routines (ISRs).

Appendix H

I/O Sample Programs

```

10  ! THIS PROGRAM OUTPUTS A STRING USING HANDSHAKE TO A GPIO-LIKE INTERFACE.
20  !
30  ! INTERFACE CARDS APPLICABLE ARE:
31  !
40  !      98032    16 BIT PARALLEL
50  !      98035    REAL TIME CLOCK
60  !      98036    SERIAL INTERFACE
70  !
80  ICOM 1000
90  DIM Input$(160)                ! ALLOW FOR 160 CHARACTER STRING
100 INTEGER Select_code           ! BASIC VARIABLE TO HOLD THE SELECT CODE
110 IASSEMBLE
120 INPUT "SELECT CODE TO WRITE TO?",Select_code
130 !
140 Input: LINPUT "STRING TO WRITE?",Input$ ! ASK USER FOR STRING TO OUTPUT
150 ICALL Output_gpio_hs(Select_code,Input$)
160 GOTO Input
170 !
180      ISOURCE      NAM Output_gpio_hs
190      ISOURCE      EXT Get_value,Error_exit
200      ISOURCE Select_code:BSS 1          ! RESERVED TO HOLD SELECT CODE
210      ISOURCE String: BSS 81             ! RESERVED FOR 160 CHAR STRING
220      ISOURCE Cr:    EQU 13              ! EQUATES FOR CR/LF
230      ISOURCE Lf:    EQU 10
240      ISOURCE !
250      ISOURCE ! ROUTINE TO OUTPUT A STRING FOLLOWED BY CR/LF TO A GPIO-LIKE
260      ISOURCE ! INTERFACE USING HANDSHAKE.
270      ISOURCE !
280      ISOURCE ! ENTRY POINT:  OUTPUT_gpio_hs
290      ISOURCE !
300      ISOURCE ! PARAMETERS:  1)  INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
310      ISOURCE !                  2)  STRING TO BE OUTPUT
320      ISOURCE !
330      ISOURCE ! POSSIBLE ERRORS:  19  SELECT CODE OUT OF RANGE
340      ISOURCE !                  164  CARD OR PERIPHERAL DOWN
350      ISOURCE !
360      ISOURCE      SUB
370      ISOURCE Parm_sc:  INT
380      ISOURCE Parm_str: STR
390      ISOURCE Output_gpio_hs: LDA =Select_code ! GET THE SELECT CODE PARM
400      ISOURCE      LDB =Parm_sc
410      ISOURCE      JSM Get_value
420      ISOURCE      LDA Select_code          ! COPY TO PA
430      ISOURCE      STA Pa
440      ISOURCE      ADA #-1                  ! CHECK FOR VALID RANGE (1-14)
450      ISOURCE      SAM Sc_error
460      ISOURCE      ADA #-15+1
470      ISOURCE      SAM Sc_ok
480      ISOURCE Sc_error: LDA =19             ! GIVE ERROR 19 IF SELECT_CODE
490      ISOURCE      JSM Error_exit          ! IS OUT OF RANGE
500      ISOURCE !

```

```

510      ISOURCE Sc_ok:      LDA =String      ! GET THE STRING PARAMETER
520      ISOURCE            LDB =Parm_str
530      ISOURCE            JSM Get_value
540      ISOURCE            LDA =String+1      ! SET UP C TO GET BYTES FROM
550      ISOURCE            SAL 1              ! THE STRING
560      ISOURCE            STA C
570      ISOURCE            CBL
580      ISOURCE            LDA String        ! IF THE STRING LENGTH IS ZERO
590      ISOURCE            SZA Done          ! THEN THERE IS NOTHING TO DO.
600      ISOURCE Write_loop: WBC A,I         ! GET THE NEXT CHAR FOR OUTPUT
610      ISOURCE            JSM Write_byte    ! OUTPUT THE CHARACTER TO CARD
620      ISOURCE            DSZ String        ! SEE IF DONE
630      ISOURCE            JMP Write_loop    ! IF NOT, REPEAT
640      ISOURCE Done:      LDA =Cr          ! NOW OUTPUT CR/LF
650      ISOURCE            JSM Write_byte
660      ISOURCE            LDA =Lf
670      ISOURCE            JSM Write_byte
680      ISOURCE            RET 1              ! RETURN TO BASIC
690      ISOURCE !
700      ISOURCE ! SUBROUTINE TO OUTPUT ONE CHARACTER TO GPIO-LIKE CARD.
710      ISOURCE !   CHARACTER IS PASSED IN A
720      ISOURCE !
730      ISOURCE Write_byte: SSC Card_down    ! SKIP IF CARD IS DOWN
740      ISOURCE            SFC Write_byte    ! ELSE WAIT FOR CARD
750      ISOURCE            STA R4            ! OUTPUT DATA TO CARD
760      ISOURCE            STA R7            ! TRIGGER HANDSHAKE
770      ISOURCE            RET 1
780      ISOURCE !
790      ISOURCE Card_down: LDA =164          ! RETURN ERROR 164 TO BASIC
800      ISOURCE            JSM Error_exit
810      ISOURCE !
820      ISOURCE            END Output_gpio_hs

```

```

10  ! THIS PROGRAM INPUTS A STRING USING HANDSHAKE FROM A GPIO-LIKE DEVICE.
20  !
30  ! INTERFACE CARDS APPLICABLE ARE:
40  !      98032    16 BIT PARALLEL
50  !      98033    BCD
60  !      98035    REAL TIME CLOCK
70  !      98036    SERIAL INTERFACE
80  !
90  ICOM 200
100 DIM Input$[160]                ! ALLOW FOR 160 CHARACTER STRING
110 INTEGER Select_code            ! BASIC VARIABLE TO HOLD THE SELECT CODE
120 IASSEMBLE
130 INPUT "SELECT CODE TO READ FROM?",Select_code
140 !
150 ICALL Read_gpio(Select_code,Input$)
160 PRINT "STRING READ=";Input$
170 END
180 !
190      ISOURCE      NAM GPIO_input
200      ISOURCE      EXT Get_value,Put_value,Error_exit
210      ISOURCE Select_code:BSS 1      ! RESERVED TO HOLD SELECT CODE
220      ISOURCE String: BSS 81          ! RESERVED FOR 160 CHAR STRING
230      ISOURCE Cr:    EQU 13           ! EQUATES FOR CR/LF
240      ISOURCE Lf:    EQU 10
250      ISOURCE !
260      ISOURCE ! ROUTINE TO INPUT A STRING FOLLOWED BY LF FROM A GPIO-LIKE
270      ISOURCE ! INTERFACE.
280      ISOURCE ! A MAX OF 160 CHARACTERS WILL BE READ. CR'S ARE IGNORED.
290      ISOURCE !
300      ISOURCE ! ENTRY POINT: READ_gpio
310      ISOURCE !
320      ISOURCE ! PARAMETERS:  1)  INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
330      ISOURCE !                  2)  STRING TO HOLD RESULT
340      ISOURCE !
350      ISOURCE ! POSSIBLE ERRORS:  19  SELECT CODE OUT OF RANGE
360      ISOURCE !                  164  CARD OR PERIPHERAL DOWN
370      ISOURCE !
380      ISOURCE      SUB
390      ISOURCE Parm_sc:  INT
400      ISOURCE Parm_str: STR
410      ISOURCE Read_gpio: LDA #Select_code    ! GET THE SELECT CODE PARAM
420      ISOURCE      LDB #Parm_sc
430      ISOURCE      JSM Get_value
440      ISOURCE      LDA Select_code          ! COPY TO PA
450      ISOURCE      STA Pa
460      ISOURCE      ADA #-1                  ! CHECK FOR VALID RANGE (1-14)
470      ISOURCE      SAM Sc_error
480      ISOURCE      ADA #-15+1
490      ISOURCE      SAM Sc_ok
500      ISOURCE Sc_error: LDA #19              ! GIVE ERROR 19 IF SELECT_CODE
510      ISOURCE      JSM Error_exit          ! IS OUT OF RANGE
520      ISOURCE !
530      ISOURCE Sc_ok:  LDA #0                ! INITIALIZE THE STRING LENGTH
540      ISOURCE      STA String
550      ISOURCE      LDA #String              ! SET UP C TO PUT BYTES INTO
560      ISOURCE      SAL 1                    ! THE STRING
570      ISOURCE      ADA #1
580      ISOURCE      STA C
590      ISOURCE      CBL
600      ISOURCE      SSC Card_down            ! SKIP IF CARD/PERIPH ARE DOWN
610      ISOURCE      SFC #-1                  ! ELSE WAIT FOR CARD

```

```

620      ISOURCE      LDA R4          ! SIGNAL THIS IS AN INPUT
630      ISOURCE Read_loop: STA R7    ! TRIGGER THE INPUT HANDSHAKE
640      ISOURCE      SFC *          ! WAIT FOR CARD TO COMPLETE
650      ISOURCE      LDA R4          ! THEN GET THE BYTE
660      ISOURCE      CPA =Lf        ! IF LINE FEED
670      ISOURCE      JMP Done       ! THEN WE ARE DONE
680      ISOURCE      CPA =Cr        ! IF CARRIAGE RETURN
690      ISOURCE      JMP Read_loop  ! THEN IGNORE IT
700      ISOURCE      PBC A,I        ! ELSE PUT CHARACTER IN STRING
710      ISOURCE      LDA String     ! AND BUMP STRING LENGTH
720      ISOURCE      ADA =1
730      ISOURCE      STA String
740      ISOURCE      CPA =160       ! HAVE WE INPUT 160 CHARS?
750      ISOURCE      JMP Done       ! YES! SO QUIT NOW
760      ISOURCE      JMP Read_loop  ! IF NOT THEN REPEAT
770      ISOURCE Done:  LDA =String  ! SEND THE STRING TO BASIC
780      ISOURCE      LDB =Parm_str
790      ISOURCE      JSM Put_value
800      ISOURCE      RET 1          ! RETURN TO BASIC
810      ISOURCE !
820      ISOURCE Card_down: LDA =164 ! RETURN ERROR 164 TO BASIC
830      ISOURCE      JSM Error_exit
840      ISOURCE !
850      ISOURCE      END Gpio_input

```

```

10  ! THIS PROGRAM OUTPUTS A STRING USING INTERRUPT TO A GPIO-LIKE INTERFACE.
20  !
30  ! INTERFACE CARDS APPLICABLE ARE:
40  !
50  !     98032   16 BIT PARALLEL
60  !     98036   SERIAL INTERFACE (INTERRUPT ENABLE BYTE SHOULD BE CHANGED)
70  !
80  ICOM 1000
90  DIM Input$(160)           ! ALLOW FOR 160 CHARACTER STRING
100 INTEGER Select_code      ! BASIC VARIABLE TO HOLD THE SELECT CODE
110 !ASSEMBLE
120 INPUT "SELECT CODE TO WRITE TO?",Select_code
130 ON INT #Select_code GOTO Isr_done ! SET UP END OF LINE BRANCH
140 !
150 Input:  INPUT "STRING TO WRITE?",Input$ ! ASK USER FOR STRING TO OUTPUT
160 ICALL Output_gpio_int(Select_code,Input$)
161 !
170 DISP I                   ! DO OTHER WORK WHILE INTERRUPT
180 I=I+1                   ! OUTPUT IS IN PROGRESS
190 GOTO 170
191 !
200 Isr_done: DISP " OUTPUT COMPLETE...NEXT "; ! GET HERE WHEN ISR OUTPUT IS
210 GOTO Input               ! COMPLETE...SO REPEAT
220 !
230      ISOURCE              NAM Output_gpio_int
240      ISOURCE              EXT Get_value,Error_exit,Isr_access
250      ISOURCE Select_code:BSS 1           ! RESERVED TO HOLD SELECT CODE
260      ISOURCE String:      BSS 81         ! RESERVED FOR 160 CHAR STRING
270      ISOURCE Byte_pointer:BSS 1         ! BYTE POINTER FOR ISR
280      ISOURCE Eol_mask:    BSS 1         ! TEMP FOR ISR
290      ISOURCE Save35:      BSS 1         ! TEMP FOR ISR
300      ISOURCE Cr:          EQU 13        ! EQUATES FOR CR/LF
310      ISOURCE Lf:          EQU 10
320      ISOURCE Enable_mask:EQU 200B      ! 98032 INTERRUPT ENABLE MASK
330      ISOURCE !
340      ISOURCE ! ROUTINE TO OUTPUT A STRING FOLLOWED BY CR/LF TO A GPIO-LIKE
350      ISOURCE ! INTERFACE USING INTERRUPT.
360      ISOURCE !
370      ISOURCE ! ENTRY POINT:  Output_gpio_int
380      ISOURCE !
390      ISOURCE ! PARAMETERS:  1)  INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
400      ISOURCE !                    2)  STRING TO BE OUTPUT
410      ISOURCE !
420      ISOURCE ! POSSIBLE ERRORS:  19  SELECT CODE OUT OF RANGE
430      ISOURCE !                    164  CARD OR PERIPHERAL DOWN
440      ISOURCE !
450      ISOURCE              SUB
460      ISOURCE Parm_sc:      INT
470      ISOURCE Parm_str:     STR
480      ISOURCE Output_gpio_int:LDA #Select_code ! GET THE SELECT CODE PARM
490      ISOURCE              LDB #Parm_sc
500      ISOURCE              JSM Get_value
510      ISOURCE              LDA Select_code    ! LOAD A WITH SELECT CODE
520      ISOURCE              ADA #-1           ! CHECK FOR VALID RANGE (1-14)
530      ISOURCE              SAM Sc_error
540      ISOURCE              ADA #-15+1
550      ISOURCE              SAM Sc_ok
560      ISOURCE Sc_error:     LDA #19          ! GIVE ERROR 19 IF SELECT_CODE
570      ISOURCE              JSM Error_exit    ! IS OUT OF RANGE
580      ISOURCE !
590      ISOURCE Sc_ok;        LDA Select_code  ! SEE IF CARD IS OK
600      ISOURCE              STA Pa           ! FIRST COPY SELECT CODE TO PA

```



```

610      ISOURCE      SSC Card_down      ! SKIP IF DOWN
620      ISOURCE      LDA =Isr          ! SET UP AN ISR
630      ISOURCE      LDB =(10*256)+(1*16)! 10 TRIES, RESOURCE=1=ASYNC
640      ISOURCE      ADB Select_code
650      ISOURCE      JSM Isr_access
660      ISOURCE      JMP Sc_ok          ! IF COULDN'T GET IT, RETRY
670      ISOURCE      LDA =String       ! GET THE STRING PARAMETER
680      ISOURCE      LDB =Parm_str
690      ISOURCE      JSM Get_value
700      ISOURCE      LDA =String+1     ! SET UP BYTE POINTER FOR ISR
710      ISOURCE      SAL 1             ! TO GET CHARS FROM STRING
720      ISOURCE      STA Byte_pointer
730      ISOURCE      ADA String        ! ADD CR/LF TO END OF STRING
740      ISOURCE      ADA =-1
750      ISOURCE      STA C
760      ISOURCE      CBL
770      ISOURCE      LDA =Cr
780      ISOURCE      PBC A,I
790      ISOURCE      LDA =Lf
800      ISOURCE      PBC A,I
810      ISOURCE      LDA String        ! BE SURE AND ADD 2 TO LENGTH
820      ISOURCE      ADA =2            ! SO ISR WILL OUTPUT CR/LF
830      ISOURCE      STA String
840      ISOURCE      LDA =Enable_mask  ! ENABLE THE CARD TO INTERRUPT
850      ISOURCE      STA R5
860      ISOURCE      RET 1             ! GO BACK TO BASIC.
870      ISOURCE      !
880      ISOURCE      Card_down: LDA =164
890      ISOURCE      JSM Error_exit
900      ISOURCE      !
910      ISOURCE      Isr: LDA 35B      ! SINCE I AM GOING TO DO STACK
920      ISOURCE      STA Save35        ! OPERATIONS, I MUST SAVE 35
930      ISOURCE      LDA 34B          ! AND INITIALIZE IT
940      ISOURCE      STA 35B
950      ISOURCE      LDA Byte_pointer  ! SET UP THE BYTE POINTER
960      ISOURCE      STA C             ! SO I CAN GET A DATA BYTE
970      ISOURCE      CBL
980      ISOURCE      WBC R4,I          ! SEND THE DATA BYTE TO CARD
990      ISOURCE      STA R7           ! DO HANDSHAKE
1000     ISOURCE      LDA C             ! RESAVE BYTE POINTER
1010     ISOURCE      STA Byte_pointer
1020     ISOURCE      DSZ String        ! SEE IF DONE
1030     ISOURCE      JMP Exit         ! IF NOT, THEN EXIT THE ISR
1040     ISOURCE      LDA =0           ! DISABLE THE CARD
1050     ISOURCE      STA R5
1060     ISOURCE      LDA Pa            ! DEPENDING ON WHETHER THE
1070     ISOURCE      ADA =-8           ! SELECT CODE IS HIGH, OR LOW
1080     ISOURCE      SAP **+3          ! CALL THE CORRECT TERMINATION
1090     ISOURCE      JSM End_isr_low,I ! ROUTINE
1100     ISOURCE      JMP **+2
1110     ISOURCE      JSM End_isr_high,I
1120     ISOURCE      LDA Pa            ! AND NOW TRIGGER AN END OF
1130     ISOURCE      ADA =-1          ! LINE BRANCH. TO DO THIS, THE
1140     ISOURCE      IOR Sb11          ! CORRECT MASK WORD MUST BE
1150     ISOURCE      LDB =1            ! CALCULATED BY A COMPUTED
1160     ISOURCE      EXE A             ! SHIFT INSTRUCTION
1170     ISOURCE      STB Eol_mask      ! SAVE THIS MASK
1180     ISOURCE      LDB Isr_psw      ! AND USE MAGIC CODE TO
1190     ISOURCE      LDA =103B        ! TRIGGER THE EOL BRANCH
1200     ISOURCE      STA B,I
1210     ISOURCE      ADB =3

```

```

1220      ISOURCE      LDA Eol_mask
1230      ISOURCE      DIR
1240      ISOURCE      IOR B,I
1250      ISOURCE      STA B,I
1260      ISOURCE      EIR
1270      ISOURCE      STA Isr_flag,I
1280      ISOURCE Exit: LDA Save35      ! RESTORE 35
1290      ISOURCE      STA 35B
1300      ISOURCE      RET 1           ! RETURN FROM INTERRUPT
1310      ISOURCE Sbit: SBL 1         ! BIT MASK FOR INSTRUCTION
1320      ISOURCE !

```

```

10  ! THIS PROGRAM INPUTS A STRING USING INTERRUPT FROM A GPIO-LIKE INTERFACE.
20  !
30  ! INTERFACE CARDS APPLICABLE ARE:
40  !
50  !     98032    16 BIT PARALLEL
60  !     98033    BCD
70  !     98036    SERIAL INTERFACE (INTERRUPT ENABLE BYTE SHOULD BE CHANGED)
80  !
90  ICOM 1000
100 DIM Input$(160)           ! ALLOW FOR 160 CHARACTER STRING
110 INTEGER Select_code       ! BASIC VARIABLE TO HOLD THE SELECT CODE
120 IASSEMBLE
130 INPUT "SELECT CODE TO READ FROM?",Select_code
140 ON INT #Select_code GOTO Isr_done ! SET UP END OF LINE BRANCH
150 ICALL Enter_gpio_int(Select_code) ! START THE READ OPERATION
160 !
170 ICALL Read_result(Input$)       ! WHILE WAITING FOR IT TO COMPLETE,
180 DISP "PARTIAL RESULT=";Input$   ! DISPLAY THE PARTIAL RESULTS
190 GOTO 170
200 !
210 Isr_done: ICALL Read_result(Input$)
220 DISP " INPUT COMPLETE...STRING=";Input$
230 END
240 !
250 ISOURCE          NAM Enter_gpio_int
260 ISOURCE          EXT Get_value,Put_value,Error_exit,Isr_access
270 ISOURCE Select_code:BSS 1       ! RESERVED TO HOLD SELECT CODE
280 ISOURCE String:   BSS 81        ! RESERVED FOR 160 CHAR STRING
290 ISOURCE Byte_pointer:BSS 1      ! BYTE POINTER FOR ISR
300 ISOURCE Eol_mask: BSS 1         ! TEMP FOR ISR
310 ISOURCE Save35:   BSS 1         ! TEMP FOR ISR
320 ISOURCE Cr:       EQU 13        ! EQUATES FOR CR/LF
330 ISOURCE Lf:       EQU 10
340 ISOURCE Enable_mask:EQU 200B    ! 98032 INTERRUPT ENABLE MASK
350 ISOURCE !
360 ISOURCE ! ROUTINES TO INPUT A STRING FOLLOWED BY LF FROM A GPIO-LIKE
370 ISOURCE ! INTERFACE USING INTERRUPT.
380 ISOURCE !
390 ISOURCE ! ENTRY POINT: Enter_gpio_int
400 ISOURCE !
410 ISOURCE ! PARAMETER: 1) INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
420 ISOURCE !
430 ISOURCE ! POSSIBLE ERRORS: 19 SELECT CODE OUT OF RANGE
440 ISOURCE !                      164 CARD OR PERIPHERAL DOWN
450 ISOURCE !
460 ISOURCE ! ENTRY POINT: Read_result
470 ISOURCE !
480 ISOURCE ! PARAMETER: 1) STRING TO CONTAIN THE INPUT DATA
490 ISOURCE !
500 ISOURCE          SUB
510 ISOURCE Parm_sc:  INT
520 ISOURCE Enter_gpio_int: LDA #Select_code ! GET THE SELECT CODE PARM
530 ISOURCE          LDB #Parm_sc
540 ISOURCE          JSM Get_value
550 ISOURCE          LDA Select_code       ! LOAD A WITH SELECT CODE
560 ISOURCE          ADA #-1               ! CHECK FOR VALID RANGE (1-14)
570 ISOURCE          SAM Sc_error
580 ISOURCE          ADA #-15+1
590 ISOURCE          SAM Sc_ok
600 ISOURCE Sc_error:  LDA #19           ! GIVE ERROR 19 IF SELECT_CODE
610 ISOURCE          JSM Error_exit      ! IS OUT OF RANGE
620 ISOURCE !

```

```

630      ISOURCE Sc_ok:      LDA Select_code      ! SEE IF CARD IS OK
640      ISOURCE            STA Pa                ! FIRST COPY SELECT CODE TO PA
650      ISOURCE            SSC Card_down         ! SKIP IF DOWN
660      ISOURCE            LDA =Isr             ! SET UP AN ISR
670      ISOURCE            LDB =(10*256)+(1*16) ! 10 TRIES, RESOURCE=1=ASYN
680      ISOURCE            ADB Select_code
690      ISOURCE            JSM Isr_access
700      ISOURCE            JMP Sc_ok             ! IF COULDN'T GET IT, RETRY
710      ISOURCE            LDA =0               ! INITIALIZE BYTE COUNT OF
720      ISOURCE            STA String           ! STRING BUFFER AREA
730      ISOURCE            LDA =String          ! SET UP BYTE POINTER FOR ISR
740      ISOURCE            SAL 1                ! TO PUT CHARS INTO STRING
750      ISOURCE            ADA =1
760      ISOURCE            STA Byte_pointer
770      ISOURCE            SFC *                ! WAIT FOR CARD
780      ISOURCE            LDA R4               ! START FIRST INPUT OPERATION
790      ISOURCE            STA R7
800      ISOURCE            LDA =Enable_mask     ! ENABLE THE CARD TO INTERRUPT
810      ISOURCE            STA R5
820      ISOURCE            RET 1                ! GO BACK TO BASIC.
830      ISOURCE !
840      ISOURCE Card_down: LDA =164
850      ISOURCE            JSM Error_exit
860      ISOURCE !
870      ISOURCE            SUB
880      ISOURCE Parm_str:  STR
890      ISOURCE Read_result: LDA =String
900      ISOURCE            LDB =Parm_str
910      ISOURCE            JSM Put_value
920      ISOURCE            RET 1
930      ISOURCE !
940      ISOURCE Isr:      LDA 35B              ! SINCE I AM GOING TO DO STACK
950      ISOURCE            STA Save35           ! OPERATIONS, I MUST SAVE 35
960      ISOURCE            LDA 34B              ! AND INITIALIZE IT
970      ISOURCE            STA 35B
980      ISOURCE            LDA Byte_pointer     ! SET UP THE BYTE POINTER
990      ISOURCE            STA C                ! SO I CAN PUT A DATA BYTE
1000     ISOURCE           CBL                  ! INTO THE STRING
1010     ISOURCE           LDA R4               ! GET THE NEXT CHARACTER FROM
1020     ISOURCE           CPA =Cr              ! THEN CARD...IGNORE CR'S
1030     ISOURCE           JMP Do_another
1040     ISOURCE           CPA =Lf              ! IF LINE FEED, THE TERMINATE
1050     ISOURCE           JMP Terminate        ! THE ISR TRANSFER
1060     ISOURCE           PBC A,I              ! ELSE PUT CHARACTER IN STRING
1070     ISOURCE           LDA C                ! SAVE NEW BYTE POINTER
1080     ISOURCE           STA Byte_pointer
1090     ISOURCE           LDA String           ! AND BUMP STRING LENGTH
1100     ISOURCE           ADA =1
1110     ISOURCE           STA String
1120     ISOURCE           CPA =160             ! HAVE WE RECEIVED 160 CHARS
1130     ISOURCE           JMP Terminate        ! IF YES, THEN SHUT DOWN
1140     ISOURCE Do_another: STA R7             ! START ANOTHER HANDSHAKE
1150     ISOURCE           JMP Exit              ! THEN EXIT THE ISR
1160     ISOURCE !
1170     ISOURCE Terminate: LDA =0              ! DISABLE THE CARD
1180     ISOURCE           STA R5
1190     ISOURCE           LDA Pa                ! DEPENDING ON WHETHER THE
1200     ISOURCE           ADA =-8              ! SELECT CODE IS HIGH, OR LOW
1210     ISOURCE           SAP **3              ! CALL THE CORRECT TERMINATION
1220     ISOURCE           JSM End_isr_low,I     ! ROUTINE
1230     ISOURCE           JMP **2
1240     ISOURCE           JSM End_isr_high,I

```

```

1250      ISOURCE      LDA Pa          ! AND NOW TRIGGER AN END OF
1260      ISOURCE      ADA #-1        ! LINE BRANCH. TO DO THIS, THE
1270      ISOURCE      IOR Sb11      ! CORRECT MASK WORD MUST BE
1280      ISOURCE      LDB #1         ! CALCULATED BY A COMPUTED
1290      ISOURCE      EXE A          ! SHIFT INSTRUCTION
1300      ISOURCE      STB Eol_mask   ! SAVE THIS MASK
1310      ISOURCE      LDB Isr_psw    ! AND USE MAGIC CODE TO
1320      ISOURCE      LDA #103B      ! TRIGGER THE EOL BRANCH
1330      ISOURCE      STA B,I
1340      ISOURCE      ADB #3
1350      ISOURCE      LDA Eol_mask
1360      ISOURCE      DIR
1370      ISOURCE      IOR B,I
1380      ISOURCE      STA B,I
1390      ISOURCE      EIR
1400      ISOURCE      STA Isr_flag,I
1410      ISOURCE Exit:  LDA Save35    ! RESTORE 35
1420      ISOURCE      STA 35B
1430      ISOURCE      RET 1          ! RETURN FROM INTERRUPT
1440      ISOURCE Sb11:  SBL 1        ! BIT MASK FOR INSTRUCTION
1450      ISOURCE      !
1460      ISOURCE      END Enter_gpio_int

```

```

10  ! THIS PROGRAM OUTPUTS A STRING USING DMA TO A GPIO INTERFACE.
20  !
30  ! INTERFACE CARDS APPLICABLE ARE:
40  !
50  !     98032   16 BIT PARALLEL
60  !
70  ICOM 1000
80  DIM Input$(160)           ! ALLOW FOR 160 CHARACTER STRING
90  INTEGER Select_code       ! BASIC VARIABLE TO HOLD THE SELECT CODE
100 IASSEMBLE
110 INPUT "SELECT CODE TO WRITE TO?",Select_code
120 ON INT #Select_code GOTO Dma_done ! SET UP END OF LINE BRANCH
130 !
140 Input: LINPUT "STRING TO WRITE?",Input$ ! ASK USER FOR STRING TO OUTPUT
150 ICALL Output_gpio_dma(Select_code,Input$)
160 !
170 DISP I                    ! DO OTHER WORK WHILE INTERRUPT
180 I=I+1                     ! OUTPUT IS IN PROGRESS
190 GOTO 170
200 !
210 Dma_done: DISP " OUTPUT COMPLETE...NEXT "; ! GET HERE WHEN ISR OUTPUT IS
220 GOTO Input                ! COMPLETE...SO REPEAT
230 !
240     ISOURCE                NAM Output_gpio_dma
250     ISOURCE                EXT Get_value,Error_exit,Isr_access
260     ISOURCE Select_code:BSS 1 ! RESERVED TO HOLD SELECT CODE
270     ISOURCE String: BSS 81 ! RESERVED FOR 160 CHAR STRING
280     ISOURCE BSS 80 ! RESERVED TO EXPAND STRING
290     ISOURCE Eol_mask: BSS 1 ! TEMP FOR ISR
300     ISOURCE Save35: BSS 1 ! TEMP FOR ISR
310     ISOURCE Cr: EQU 13 ! EQUATES FOR CR/LF
320     ISOURCE Lf: EQU 10
330     ISOURCE Enable_mask:EQU 320B ! 98032 DMA/INT/AH ENABLE MASK
340     ISOURCE !
350     ISOURCE ! ROUTINE TO OUTPUT A STRING FOLLOWED BY CR/LF TO A GPIO-LIKE
360     ISOURCE ! INTERFACE USING DMA.
370     ISOURCE !
380     ISOURCE ! ENTRY POINT: Output_gpio_dma
390     ISOURCE !
400     ISOURCE ! PARAMETERS: 1) INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
410     ISOURCE ! 2) STRING TO BE OUTPUT
420     ISOURCE !
430     ISOURCE ! POSSIBLE ERRORS: 19 SELECT CODE OUT OF RANGE
440     ISOURCE ! 164 CARD OR PERIPHERAL DOWN
450     ISOURCE !
460     ISOURCE SUB
470     ISOURCE Parm_sc: INT
480     ISOURCE Parm_str: STR
490     ISOURCE Output_gpio_dma:LDA =Select_code ! GET THE SELECT CODE PARM
500     ISOURCE LDB =Parm_sc
510     ISOURCE JSM Get_value
520     ISOURCE LDA Select_code ! LOAD A WITH SELECT CODE
530     ISOURCE ADA =-1 ! CHECK FOR VALID RANGE (1-14)
540     ISOURCE SAM Sc_error
550     ISOURCE ADA =-15+1
560     ISOURCE SAM Sc_ok
570     ISOURCE Sc_error: LDA =19 ! GIVE ERROR 19 IF SELECT_CODE
580     ISOURCE JSM Error_exit ! IS OUT OF RANGE
590     ISOURCE !
600     ISOURCE Sc_ok: LDA Select_code ! SEE IF CARD IS OK
610     ISOURCE STA Pa ! FIRST COPY SELECT CODE TO PA
620     ISOURCE SSS Card_ok ! SKIP IF CARD IS UP

```

```

630      ISOURCE      LDA =164          ! ELSE GIVE ERROR 164
640      ISOURCE      JSM Error_exit
650      ISOURCE      !
660      ISOURCE      Card_ok:      LDA =Isr          ! SET UP AN ISR
670      ISOURCE      LDB =(10*256)+(2*16)! 10 TRIES, RESOURCE=2=DMA
680      ISOURCE      ADB Select_code
690      ISOURCE      JSM Isr_access
700      ISOURCE      JMP Sc_ok        ! IF COULDN'T GET IT, RETRY
710      ISOURCE      LDA =String      ! GET THE STRING PARAMETER
720      ISOURCE      LDB =Parm_str
730      ISOURCE      JSM Get_value
740      ISOURCE      ! FOR DMA, THE NORMAL STRING FORMAT WON'T DO. THE DATA MUST
750      ISOURCE      ! BE STORED ONE BYTE PER WORD, SO THE FOLLOWING LOOP WILL
760      ISOURCE      ! EXPAND THE STRING AND ADD A CR/LF
770      ISOURCE      LDA =String+1    ! FIRST SET UP BYTE POINTER TO
780      ISOURCE      SAL 1              ! WITHDRAW THE LAST CHARACTER
790      ISOURCE      ADA String        ! FIRST
800      ISOURCE      ADA =-1
810      ISOURCE      STA C              ! USE C FOR THE BYTE POINTER
820      ISOURCE      CBL
830      ISOURCE      LDA =String+3    ! NOW COMPUTE A WORD POINTER
840      ISOURCE      ADA String        ! TO WHERE TO PLACE THE LF
850      ISOURCE      STA D
860      ISOURCE      LDA =Lf           ! MOVE IN A LF
870      ISOURCE      PWD A,D
880      ISOURCE      LDA =Cr          ! AND CR
890      ISOURCE      PWD A,D
900      ISOURCE      LDB String        ! NOW LOOP TO COPY ALL BYTES
910      ISOURCE      TCB
920      ISOURCE      SIB **4
930      ISOURCE      WBC A,D
940      ISOURCE      PWD A,D
950      ISOURCE      RIB **2
960      ISOURCE      LDA String        ! SET UP DMA CONTROL REGISTERS
970      ISOURCE      ADA =1            ! COUNT = #CHARS-1
980      ISOURCE      STA Dmac
990      ISOURCE      LDA =String+1    ! DMA = DATA ADDRESS
1000     ISOURCE      STA Dmama
1010     ISOURCE      SDO              ! SET DMA OUTWARDS
1020     ISOURCE      LDA =Enable_mask ! ENABLE THE CARD TO INTERRUPT
1030     ISOURCE      STA R5
1040     ISOURCE      DMA
1050     ISOURCE      RET 1              ! GO BACK TO BASIC.
1060     ISOURCE      !
1070     ISOURCE      Isr:      LDA 35B   ! I WILL GET THE INTERRUPT
1080     ISOURCE      STA Save35        ! THE DMA TRANSFER IS COMPLETE
1090     ISOURCE      LDA 34B
1100     ISOURCE      STA 35B
1110     ISOURCE      LDA =0           ! SO DISABLE THE CARD
1120     ISOURCE      STA R5
1130     ISOURCE      DDR              ! DISABLE DMA
1140     ISOURCE      LDA Pa            ! DEPENDING ON WHETHER THE
1150     ISOURCE      ADA =-8           ! SELECT CODE IS HIGH, OR LOW
1160     ISOURCE      SAP **3          ! CALL THE CORRECT TERMINATION
1170     ISOURCE      JSM End_isr_low,I ! ROUTINE
1180     ISOURCE      JMP **2
1190     ISOURCE      JSM End_isr_high,I
1200     ISOURCE      LDA Pa            ! AND NOW TRIGGER AN END OF
1210     ISOURCE      ADA =-1           ! LINE BRANCH. TO DO THIS, THE
1220     ISOURCE      IOR Sb11          ! CORRECT MASK WORD MUST BE
1230     ISOURCE      LDB =1            ! CALCULATED BY A COMPUTED
1240     ISOURCE      EXE A             ! SHIFT INSTRUCTION

```

```

1250      ISOURCE      STB Eol_mask      ! SAVE THIS MASK
1260      ISOURCE      LDB Isr_psw      ! AND USE MAGIC CODE TO
1270      ISOURCE      LDA #103B        ! TRIGGER THE EOL BRANCH
1280      ISOURCE      STA B,I
1290      ISOURCE      ADB #3
1300      ISOURCE      LDA Eol_mask
1310      ISOURCE      DIR
1320      ISOURCE      IOR B,I
1330      ISOURCE      STA B,I
1340      ISOURCE      EIR
1350      ISOURCE      STA Isr_flag,I
1360      ISOURCE      LDA Save35        ! RESTORE 35
1370      ISOURCE      STA 35B
1380      ISOURCE      RET 1             ! RETURN FROM INTERRUPT
1390      ISOURCE      Sbl1:            ! BIT MASK FOR INSTRUCTION
1400      ISOURCE      !
1410      ISOURCE      END Output_gpio_dma

```



```

10  ! THIS PROGRAM INPUTS A STRING USING DMA FROM A GPIO INTERFACE.
20  !
30  ! INTERFACE CARDS APPLICABLE ARE:
40  !
50  !     98032    16 BIT PARALLEL
60  !
70  ICOM 1000
80  DIM Input$(160)           ! ALLOW FOR 160 CHARACTER STRING
90  INTEGER Select_code       ! BASIC VARIABLE TO HOLD THE SELECT CODE
100 INTEGER Character_count   ! VARIABLE TO HOLD INPUT CHARACTER COUNT
110 INTEGER A,C               ! VARIABLES FOR "BACKGROUND PROCESS"
120 ASSEMBLE
130 INPUT "SELECT CODE TO READ FROM?",Select_code
140 ON INT #Select_code GOTO Isr_done ! SET UP END OF LINE BRANCH
150 INPUT "NUMBER OF CHARACTERS TO READ?",Character_count
160 ICALL Enter_gpio_dma(Select_code,Character_count) ! START THE READ
170 !
180 ICALL Test_dma(C,A)        ! WHILE WAITING, DISPLAY DMA COUNT AND
190 DISP "DMA COUNT=";C,"ADDRESS=";A,I ! ADDRESS
200 I=I+1
210 GOTO 180
220 !
230 Isr_done: ICALL Read_result(Input$)
240 DISP " INPUT COMPLETE...STRING=";Input$
250 END
260 !
270      ISOURCE          NAM Enter_gpio_dma
280      ISOURCE          EXT Get_value,Put_value,Error_exit,Isr_access
290      ISOURCE Select_code:BSS 1          ! RESERVED TO HOLD SELECT CODE
300      ISOURCE String:  BSS 81          ! RESERVED FOR 160 CHAR STRING
310      ISOURCE          BSS 80          ! RESERVED FOR EXPANDED STRING
320      ISOURCE Eol_mask:  BSS 1          ! TEMP FOR ISR
330      ISOURCE Save35:   BSS 1          ! TEMP FOR ISR
340      ISOURCE Enable_mask:EQU 320B      ! 98032 DMA/INT/AH ENABLE MASK
350      ISOURCE !
360      ISOURCE ! ROUTINES TO INPUT A FIXED LENGTH STRING FROM A GPIO
370      ISOURCE ! INTERFACE USING DMA.
380      ISOURCE !
390      ISOURCE ! ENTRY POINT:  Enter_gpio_dma
400      ISOURCE !
410      ISOURCE ! PARAMETER:   1)  INTEGER CONTAINING SELECT CODE ( 1 TO 14 )
420      ISOURCE !              2)  NUMBER OF CHARACTERS TO READ ( 1 TO 80 )
430      ISOURCE !
440      ISOURCE ! POSSIBLE ERRORS:  19  SELECT CODE OR CHAR COUNT OUT OF RANGE
450      ISOURCE !                  164  CARD OR PERIPHERAL DOWN
460      ISOURCE !
470      ISOURCE ! ENTRY POINT:  Test_dma
480      ISOURCE !
490      ISOURCE ! PARAMETERS:  1)  INTEGER TO HOLD CURRENT DMA COUNT
500      ISOURCE !              2)  INTEGER TO HOLD CURRENT DMA ADDRESS
510      ISOURCE !
520      ISOURCE ! ENTRY POINT:  Read_result
530      ISOURCE !
540      ISOURCE ! PARAMETER:   1)  STRING TO CONTAIN THE INPUT DATA
550      ISOURCE !
560      ISOURCE          SUB
570      ISOURCE Parm_sc:  INT
580      ISOURCE Parm_count: INT
590      ISOURCE Enter_gpio_dma: LDA =Select_code ! GET THE SELECT CODE PARM
600      ISOURCE          LDB =Parm_sc
610      ISOURCE          JSM Get_value
620      ISOURCE          LDA Select_code      ! LOAD A WITH SELECT CODE

```

```

630      ISOURCE      ADA =-1          ! CHECK FOR VALID RANGE (1-14)
640      ISOURCE      SAM Sc_error
650      ISOURCE      ADA =-15+1
660      ISOURCE      SAM Sc_ok
670      ISOURCE Sc_error: LDA =19      ! GIVE ERROR 19 IF SELECT_CODE
680      ISOURCE      JSM Error_exit    ! IS OUT OF RANGE
690      ISOURCE !
700      ISOURCE Sc_ok:  LDA =String     ! GET BYTE COUNT PARAMETER
710      ISOURCE      LDB =Parm_count
720      ISOURCE      JSM Get_value
730      ISOURCE      LDA String        ! CHECK IT FOR RANGE
740      ISOURCE      SAM Sc_error
750      ISOURCE      SZA Sc_error
760      ISOURCE      ADA =-81
770      ISOURCE      SAP Sc_error
780      ISOURCE Check_card: LDA Select_code ! SEE IF CARD IS OK
790      ISOURCE      STA Pa            ! FIRST COPY SELECT CODE TO PA
800      ISOURCE      SSS Card_ok      ! SKIP IF CARD IS OK
810      ISOURCE      LDA =164         ! ELSE GIVE ERROR 164
820      ISOURCE      JSM Error_exit
830      ISOURCE !
840      ISOURCE Card_ok:  LDA =Isr      ! SET UP AN ISR
850      ISOURCE      LDB =(10*256)+(2*16)! 10 TRIES, RESOURCE=2=DMA
860      ISOURCE      ADB Select_code
870      ISOURCE      JSM Isr_access
880      ISOURCE      JMP Check_card    ! IF COULDN'T GET IT, RETRY
890      ISOURCE      LDA String        ! INITIALIZE DMA REGISTERS
900      ISOURCE      ADA =-1
910      ISOURCE      STA Dmac
920      ISOURCE      LDA =String+1
930      ISOURCE      STA Dmama
940      ISOURCE      SDI
950      ISOURCE      SFC *              ! WAIT FOR CARD
960      ISOURCE      LDA R4             ! START FIRST INPUT OPERATION
970      ISOURCE      STA R7
980      ISOURCE      LDA =Enable_mask  ! ENABLE THE CARD TO INTERRUPT
990      ISOURCE      STA R5
1000     ISOURCE      DMA
1010     ISOURCE      RET 1              ! ENABLE PROCESSOR FOR DMA
1020     ISOURCE !                      ! GO BACK TO BASIC.
1030     ISOURCE      SUB
1040     ISOURCE C_parm:  INT
1050     ISOURCE A_parm:  INT
1060     ISOURCE Test_dma: LDA Dmac
1070     ISOURCE      STA Temp
1080     ISOURCE      LDA =Temp
1090     ISOURCE      LDB =C_parm
1100     ISOURCE      JSM Put_value
1110     ISOURCE      LDA Dmama
1120     ISOURCE      STA Temp
1130     ISOURCE      LDA =Temp
1140     ISOURCE      LDB =A_parm
1150     ISOURCE      JSM Put_value
1160     ISOURCE      RET 1
1170     ISOURCE Temp:    BSS 1
1180     ISOURCE !
1190     ISOURCE      SUB
1200     ISOURCE Parm_str: STR
1210     ISOURCE Read_result: LDA =String+1 ! I MUST PACK THE STRING FROM
1220     ISOURCE      STA D              ! FROM 1 BYTE TO 2 BYTES PER
1230     ISOURCE      SAL 1
1240     ISOURCE      ADA =-1

```

```

1250      ISOURCE      STA C
1260      ISOURCE      CBL
1270      ISOURCE      LDA String          ! GET CHARACTER COUNT
1280      ISOURCE      TCA
1290      ISOURCE      SIA **4
1300      ISOURCE      MWD B,I             ! GET A BYTE
1310      ISOURCE      PBC B,I             ! PACK IT
1320      ISOURCE      RIA **2
1330      ISOURCE      LDA =String         ! RETURN RESULT TO BASIC
1340      ISOURCE      LDB =Parm_str
1350      ISOURCE      JSM Put_value
1360      ISOURCE      RET 1
1370      ISOURCE      !
1380      ISOURCE      Isr: LDA 35B          ! I WILL GET AN INTERRUPT WHEN
1390      ISOURCE          STA Save35        ! THE DMA IS COMPLETE
1400      ISOURCE      LDA 34B
1410      ISOURCE      STA 35B
1420      ISOURCE      LDA Dmac             ! I GET TO HERE WHEN DMA DONE
1430      ISOURCE      ADA =1              ! COMPUTE ACTUAL NUMBER OF
1440      ISOURCE      TCA                  ! CHARACTERS TRANSFERED
1450      ISOURCE      ADA String
1460      ISOURCE      STA String           ! SAVE IN STRING LENGTH WORD
1470      ISOURCE      LDA =0              ! DISABLE THE CARD
1480      ISOURCE      STA R5
1490      ISOURCE      DDR                  ! DISABLE DMA
1500      ISOURCE      LDA Pa               ! DEPENDING ON WHETHER THE
1510      ISOURCE      ADA =-8              ! SELECT CODE IS HIGH, OR LOW
1520      ISOURCE      SAP **3              ! CALL THE CORRECT TERMINATION
1530      ISOURCE      JSM End_isr_low,I    ! ROUTINE
1540      ISOURCE      JMP **2
1550      ISOURCE      JSM End_isr_high,I
1560      ISOURCE      LDA Pa               ! AND NOW TRIGGER AN END OF
1570      ISOURCE      ADA =-1              ! LINE BRANCH. TO DO THIS, THE
1580      ISOURCE      IOR Sb11             ! CORRECT MASK WORD MUST BE
1590      ISOURCE      LDB =1               ! CALCULATED BY A COMPUTED
1600      ISOURCE      EXE A                ! SHIFT INSTRUCTION
1610      ISOURCE      STB Eol_mask         ! SAVE THIS MASK
1620      ISOURCE      LDB Isr_psw         ! AND USE MAGIC CODE TO
1630      ISOURCE      LDA =103B           ! TRIGGER THE EOL BRANCH
1640      ISOURCE      STA B,I
1650      ISOURCE      ADB =3
1660      ISOURCE      LDA Eol_mask
1670      ISOURCE      DIR
1680      ISOURCE      IOR B,I
1690      ISOURCE      STA B,I
1700      ISOURCE      EIR
1710      ISOURCE      STA Isr_flag,I
1720      ISOURCE      LDA Save35
1730      ISOURCE      STA 35B
1740      ISOURCE      RET 1                ! RETURN FROM INTERRUPT
1750      ISOURCE      Sb11: SBL 1          ! BIT MASK FOR INSTRUCTION
1760      ISOURCE      !
1770      ISOURCE      END Enter_gpio_dma 3333333---' ,

```

```

10 ! 98034A HPiB CARD DRIVER
20 !
30 ! TWO ASSEMBLY LANGUAGE DRIVERS ARE PROVIDED...ONE FOR OUTPUT AND ONE
40 ! FOR INPUT. BOTH HAVE PROVISIONS FOR INCLUDING A BUS COMMAND STRING
50 ! FOR ADDRESSING THE BUS.
60 !
70 ! SYNTAX:
80 !
90 ! ICALL Hpib_output( <ISC>, <CMD#>, [ <DATA#> ] )
100 ! ICALL Hpib_enter( <ISC>, <CMD#>, [ <VAR#> ] )
110 !
120 ! <ISC> ::= INTERFACE SELECT CODE (1 TO 14) (INTEGER)
130 ! <CMD#> ::= STRING TO OUTPUT WITH ATN TRUE
140 ! <DATA#> ::= STRING TO OUTPUT WITH ATN FALSE
150 ! <VAR#> ::= STRING VARIABLE TO HOLD DATA READ FROM BUS
160 !
170 ! POSSIBLE ERRORS:
180 !
190 ! 164 CARD WAS NOT AN HPiB CARD
200 ! 500 <CMD#> WAS NON-NULL BUT THE CARD WAS NOT ACTIVE CONTROLLER
210 ! 501 <DATA#> WAS NON-NULL BUT THE CARD WAS NOT ACTIVER TALKER
220 ! 502 <VAR#> WAS SPECIFIED BUT THE CARD WAS NOT ACTIVE LISTENER
230 !
240 ICOM 1000
250 INTEGER Select_code
260 DIM Cmd#[160],Data#[160],Var#[160]
270 IASSEMBLE
280 INPUT "HPiB SELECT CODE?",Select_code
290 ON KEY #0 GOSUB Output
300 ON KEY #1 GOSUB Enter
310 PRINT "KEY0 = OUTPUT KEY1 = ENTER"
320 DISP "IDLE"
330 GOTO 320
340 Output: GOSUB Linput_cmd
350 LINPUT "DATA TO SEND?",Data#
360 ICALL Hpib_output(Select_code,Cmd#,Data#)
370 PRINT " DATA SENT =";Data#
380 RETURN
390 Enter: GOSUB Linput_cmd
400 ICALL Hpib_enter(Select_code,Cmd#,Var#)
410 PRINT " DATA READ =";Var#
420 RETURN
430 !
440 Linput_cmd: LINPUT "COMMAND BYTES?",Cmd#
450 RETURN
460 !
470 ISOURCE NAM Hpib
480 ISOURCE EXT Get_value,Put_value,Error_exit
490 ISOURCE Cmd: BSS 81 ! STRING TO HOLD CMD BYTES
500 ISOURCE Data: EQU Cmd ! STRING TO HOLD DATA BYTES
510 ISOURCE Select_code:BSS 1 ! INTERFACE SELECT CODE
520 ISOURCE Parm_ptr: BSS 1 ! POINTER TO PARM PSEUDO OPS
530 ISOURCE Lf: EQU 10 ! EQUATES
540 ISOURCE Cr: EQU 13
550 ISOURCE Status1: BSS 1 ! 4 WORDS TO CONTAIN STATUS
560 ISOURCE Status2: BSS 1 ! BYTES FROM 98034
570 ISOURCE Status3: BSS 1
580 ISOURCE Status4: BSS 1
590 ISOURCE !
600 ISOURCE Out_parm: SUB
610 ISOURCE , INT

```

```

620      ISOURCE      STR
630      ISOURCE P_data:  STR
640      ISOURCE Hpib_output: LDB =Out_parm      ! CALL SETUP ROUTINE
650      ISOURCE      JSM Hpib_setup
660      ISOURCE      LDA Out_parm              ! IS THERE A DATA PARAMETER?
670      ISOURCE      CPA =2
680      ISOURCE No_output:  RET 1                ! NO, RETURN TO BASIC
690      ISOURCE      LDA =Data                  ! YES, FETCH IT
700      ISOURCE      LDB =P_data
710      ISOURCE      JSM Get_value
720      ISOURCE      LDA Data                  ! CHECK BYTE COUNT
730      ISOURCE      SZA No_output              ! IF ZERO, DO NOTHING
740      ISOURCE      JSM Hpib_status            ! MAKE SURE WE ARE ADDRESSED
750      ISOURCE      LDA Status4                ! TO TALK
760      ISOURCE      AND =40B
770      ISOURCE      RZA ++3
780      ISOURCE      LDA =501                  ! ELSE GIVE ERROR 501
790      ISOURCE      JSM Error_exit
800      ISOURCE      LDA =Data+1                ! ELSE COMPUTE BYTE POINTER
810      ISOURCE      SAL 1                      ! SO WE CAN WITHDRAW BYTES
820      ISOURCE      STA C                      ! FROM THE STRING
830      ISOURCE      CBL
840      ISOURCE Data_loop: SFC *                ! WAIT FOR CARD
850      ISOURCE      WBC R4,I                  ! OUTPUT A BYTE
860      ISOURCE      DSZ Data                  ! SEE IF DONE WITH STRING
870      ISOURCE      JMP Data_loop              ! NO
880      ISOURCE      RET 1                      ! DONE, SO GO BACK TO BASIC
890      ISOURCE !
900      ISOURCE Ent_parm:  SUB
910      ISOURCE      INT
920      ISOURCE      STR
930      ISOURCE Ent_var:  STR
940      ISOURCE Hpib_enter: LDB =Ent_parm        ! CALL SETUP ROUTINE
950      ISOURCE      JSM Hpib_setup
960      ISOURCE      LDA =Ent_parm              ! IS THERE A DATA PARAMETER?
970      ISOURCE      CPA =2
980      ISOURCE      RET 1                      ! NO, THEN I'D DONE
990      ISOURCE      JSM Hpib_status            ! MAKE SURE I'M A LISTENER
1000     ISOURCE      LDA Status4
1010     ISOURCE      AND =20B
1020     ISOURCE      RZA ++3
1030     ISOURCE      LDA =502                  ! ELSE GIVE ERROR 502
1040     ISOURCE      JSM Error_exit
1050     ISOURCE      LDA =0                      ! CLEAR DATA STRING COUNTER
1060     ISOURCE      STA Data
1070     ISOURCE      LDA =Data                  ! SET UP BYTE POINTER FOR DATA
1080     ISOURCE      SAL 1
1090     ISOURCE      ADA =1
1100     ISOURCE      STA C
1110     ISOURCE      CBL
1120     ISOURCE Enter_loop: SFC *                ! WAIT FOR CARD
1130     ISOURCE      LDA R4                      ! START ACCEPTOR HANDSHAKE
1140     ISOURCE      SFC *                      ! WAIT FOR DATA
1150     ISOURCE      LDA R6                      ! READ DATA FROM CARD
1160     ISOURCE      CPA =Cr                    ! IS IT A RETURN?
1170     ISOURCE      JMP Enter_loop              ! IF SO, IGNORE IT
1180     ISOURCE      CPA =Lf                    ! IS IT TERMINATOR?
1190     ISOURCE      JMP Ent_done                ! YES, SKIP
1200     ISOURCE      PBC A,I                    ! ELSE PUT BYTE INTO STRING
1210     ISOURCE      ISZ Data                  ! BUMP STRING LENGTH
1220     ISOURCE      JMP Enter_loop              ! REPEAT FOR NEXT BYTE

```

```

1230      ISOURCE Ent_done:   LDA #Data           ! RETURN DATA TO PARAMETER
1240      ISOURCE           LDB #Ent_var
1250      ISOURCE           JSM Put_value
1260      ISOURCE           RET 1
1270      ISOURCE !
1280      ISOURCE ! HPIB SETUP ROUTINE
1290      ISOURCE !   B POINTS TO SUB PSEUDO OP (CONTAINS PARM COUNT)
1300      ISOURCE !   1) VERIFY PARAMETER COUNT >=2
1310      ISOURCE !   2) FETCH SELECT CODE AND VERIFY CARD IS A 98034A
1320      ISOURCE !   3) FETCH COMMAND STRING PARAMETER AND OUTPUT IT
1330      ISOURCE !
1340      ISOURCE Hpib_setup: LDA B,I             ! CHECK PARM COUNT
1350      ISOURCE           ADA #-2
1360      ISOURCE           SAP ++3             ! SKIP IF >=2
1370      ISOURCE           LDA #8             ! IF <2, GIVE ERROR 8
1380      ISOURCE           JSM Error_exit
1390      ISOURCE           ADB #1             ! POINT TO SELECT CODE PARM
1400      ISOURCE           STB Parm_ptr
1410      ISOURCE           LDA #Select_code    ! FETCH IT
1420      ISOURCE           JSM Get_value
1430      ISOURCE           LDA Select_code    ! CHECK RANGE FOR 1 TO 14
1440      ISOURCE           ADA #-1
1450      ISOURCE           SAM Sc_error
1460      ISOURCE           ADA #-15+1
1470      ISOURCE           SAM ++3
1480      ISOURCE Sc_error:   LDA #19           ! IF OUT OF RANGE, GIVE ERROR
1490      ISOURCE           JSM Error_exit      ! 19
1500      ISOURCE           LDA Select_code    ! SET UP PA AND DO STATUS SEQ
1510      ISOURCE           STA Pa             ! ON CARD TO VERIFY IT IS A
1520      ISOURCE           JSM Hpib_status    ! 98034A INTERFACE
1530      ISOURCE           LDB Parm_ptr      ! NOW FETCH COMMAND STRING
1540      ISOURCE           ADB #3
1550      ISOURCE           LDA #Cmd
1560      ISOURCE           JSM Get_value
1570      ISOURCE           LDA Cmd            ! SEE IF THERE IS ANYTHING
1580      ISOURCE           SZA No_cmd         ! OUTPUT, IF NOT, SKIP
1590      ISOURCE           LDA Status4       ! MAKE SURE I AM ACTIVE
1600      ISOURCE           AND #100B        ! CONTROLLER
1610      ISOURCE           RZA ++3           ! SKIP IF YES
1620      ISOURCE           LDA #500          ! ELSE GIVE ERROR 500
1630      ISOURCE           JSM Error_exit
1640      ISOURCE           LDA #Cmd+1         ! NOW OUTPUT THE COMMANDS
1650      ISOURCE           SAL 1
1660      ISOURCE           STA C
1670      ISOURCE           CBL
1680      ISOURCE Cmd_loop:   SFC *
1690      ISOURCE           WBC R6,I           ! SEND OUT CMD BYTE
1700      ISOURCE           DSZ Cmd           ! SEE IF DONE
1710      ISOURCE           JMP Cmd_loop      ! NOT YET
1720      ISOURCE No_cmd:    RET 1            ! DONE!
1730      ISOURCE !
1740      ISOURCE ! STATUS SEQUENCE FOR 98034 CARD. NOTE THAT THIS SEQUENCE
1750      ISOURCE ! COULD FORCE THE CARD TO VIOLATE THE IFC TIME SPECS IF
1760      ISOURCE ! THE FOLLOWING CONDITIONS EXIST:
1770      ISOURCE !   1) CARD IS NOT SYSTEM CONTROLLER
1780      ISOURCE !   2) A HARDWARE INTERRUPT OCCURS AFTER THE LDA R5 BUT
1790      ISOURCE !      BEFORE THE DIR
1800      ISOURCE !   3) THE CONTROLLER PULLS IFC AFTER THE LDA R5 BUT BEFORE
1810      ISOURCE !      THE DIR
1820      ISOURCE ! THE ONLY ALTERNATIVE TO THIS IS TO DIR BEFORE THE LDA R5.
1830      ISOURCE ! THIS HOWEVER COULD COMPROMISE ANY SYNCHRONOUS INTERRUPT

```

```

1840      ISOURCE ! TRANSFER IN PROGRESS ( FOR EXAMPLE THE TAPE CARTRIDGE ).
1850      ISOURCE !
1860      ISOURCE Hpib_status:SFC *           ! GET THE CARD INTO
1870      ISOURCE          LDA R5             ! IT'S STATUS SEQUENCE.
1880      ISOURCE          AND =60B           ! MAKE SURE IT IS A 98034
1890      ISOURCE          CPA =60B
1900      ISOURCE          JMP #+3           ! YES
1910      ISOURCE          LDA =164          ! IF NOT, GIVE ERROR 164
1920      ISOURCE          JSM Error_exit
1930      ISOURCE          SFC *
1940      ISOURCE          DIR               ! (THIS IS THE CRITICAL TIME)
1950      ISOURCE          SFC *             ! MADE IT, SO DISABLE MY
1960      ISOURCE          LDA R6             ! INTERRUPTS FOR THE REST OF
1970      ISOURCE          STA Status1        ! THE STATUS SEQUENCE.
1980      ISOURCE          SFC *
1990      ISOURCE          LDA R6
2000      ISOURCE          STA Status2
2010      ISOURCE          SFC *
2020      ISOURCE          LDA R6
2030      ISOURCE          STA Status3
2040      ISOURCE          SFC *
2050      ISOURCE          LDA R6
2060      ISOURCE          EIR
2070      ISOURCE          STA Status4
2080      ISOURCE          RET 1
2090      ISOURCE          END Hpib

```

```

10  ! PROGRAM TO DEMONSTRATE USING THE CLOCK FOR INTERRUPTS
20  !
30  ! THIS EXAMPLE SHOWS HOW TO USE THE CLOCK INTERRUPT TO PUT THE TIME
40  ! OF DAY INTO THE SYSTEM MESSAGE AREA AS LONG AS THE PROGRAM IS RUNNING.
50  !
60  ! THE CLOCK IS PROGRAMMED TO GENERATE AN INTERRUPT EVERY SECOND. THE
70  ! ASSEMBLY INTERRUPT SERVICE ROUTINE TRIGGERS AN END OF LINE BRANCH. THE
80  ! EOL BRANCH ROUTINE CALLS AN ASSEMBLY ROUTINE TO PUT THE TIME OF DAY
90  ! INTO THE SYSTEM MESSAGE AREA.
100 !
110 ICOM 200
120 IASSEMBLE
130 ICALL Setup_clock          ! SET UP ISR AND START CLOCK
140 ON INT #9 CALL Time        ! SET UP EOL BRANCH
150 !
160 ! BACKGROUND PROGRAM:
170 !
180 DISP I
190 I=I+1
200 GOTO 180
210 !
220 SUB Time
230 ICALL Display_time
240 SUBEXIT
250     ISOURCE      NAM Time
260     ISOURCE      EXT Error_exit,Printer_select,Print_string
270     ISOURCE      EXT Isr_access
280     ISOURCE Select_code:EQU 9
290     ISOURCE Eol_mask: SET 1          ! GET ASSEMBLER TO COMPUTE
300     ISOURCE      REP Select_code    ! THE EOL MASK FOR TRIGGERING
310     ISOURCE Eol_mask: SET Eol_mask*2 ! EOL BRANCHES
320     ISOURCE Cr: EQU 13              ! OTHER EQUATES
330     ISOURCE Lf: EQU 10
340     ISOURCE String: BSS 20          ! AREA TO HOLD TIME OF DAY
350     ISOURCE Old_pi: BSS 1           ! TWO WORDS TO HOLD CURRENT
360     ISOURCE Old_pw: BSS 1           ! PRINTER IS AND PRINTER WIDTH
370     ISOURCE !
380     ISOURCE      SUB
390     ISOURCE Setup_clock: LDA =Select_code ! MAKE SURE THE CLOCK CARD
400     ISOURCE      STA Pa              ! IS ALIVE
410     ISOURCE      SSS Card_ok
420     ISOURCE Card_down: LDA =164      ! IF NOT, GIVE ERROR 164
430     ISOURCE      JSM Error_exit
440     ISOURCE Card_ok: LDA =Isr        ! SET UP ISR LINKAGE
450     ISOURCE      LDB =(10*256)+(1*16)+Select_code
460     ISOURCE      JSM Isr_access
470     ISOURCE      JMP #+2            ! IF ERROR, THEN JUMP
480     ISOURCE      JMP Start_card     ! ELSE GO START UP THE CARD
490     ISOURCE      CPA =-1            ! IF DIDN'T GET RESOURCES
500     ISOURCE      JMP Setup_clock    ! THEN TRY AGAIN
510     ISOURCE      RET 1              ! IF ISR ALREADY LINKED, RETURN
520     ISOURCE Start_card: LDA ="U4H/U4=04/U4P1000/U4G"+Lf
530     ISOURCE      SAL 1              ! SET UP C TO POINT TO STRING
540     ISOURCE      STA C              ! WHICH I WILL OUTPUT TO THE
550     ISOURCE      CBL                ! CLOCK TO PROGRAM IT.
560     ISOURCE      LDB =-21           ! B IS -(CHAR COUNT-1)
570     ISOURCE Out_loop: SFC #         ! WAIT FOR CARD
580     ISOURCE      WBC R4,I           ! SHOVE NEXT BYTE OUT TO CARD
590     ISOURCE      STA R7             ! TRIGGER HANDSHAKE
600     ISOURCE      RIB Out_loop       ! LOOP UNTIL DONE
610     ISOURCE      LDA =200B         ! ENABLE THE CARD TO INTERRUPT

```



```

620      ISOURCE      STA R5
630      ISOURCE      RET 1
640      ISOURCE !
650      ISOURCE      SUB
660      ISOURCE Display_time: LDA =Select_code      ! FETCH TIME FROM CLOCK
670      ISOURCE      STA Pa
680      ISOURCE      SSC Card_down      ! OOPS...CARD WENT DOWN
690      ISOURCE      LDA =R      ! OUTPUT "R" TO CLOCK TO GET
700      ISOURCE      SFC *      ! IT TO GIVE ME THE TIME
710      ISOURCE      STA R4
720      ISOURCE      STA R7
730      ISOURCE      LDA =Lf
740      ISOURCE      SFC *
750      ISOURCE      STA R4
760      ISOURCE      STA R7
770      ISOURCE      LDA =String      ! SET UP C TO PUT TIME OF DAY
780      ISOURCE      SAL 1      ! DATA INTO STRING
790      ISOURCE      ADA =1
800      ISOURCE      STA C
810      ISOURCE      CBL
820      ISOURCE      LDA =0      ! CLEAR THE STRING COUNT
830      ISOURCE      STA String
840      ISOURCE      SFC *      ! WAIT FOR CARD
850      ISOURCE      LDA R4      ! START INPUT OPERATION
860      ISOURCE Read_loop: STA R7      ! TRIGGER HANDSHAKE
870      ISOURCE      SFC *      ! WAIT FOR CARD
880      ISOURCE      LDA R4      ! GET THE NEXT BYTE
890      ISOURCE      CPA =Cr      ! IGNORE CR'S
900      ISOURCE      JMP Read_loop
910      ISOURCE      CPA =Lf      ! TERMINATE ON LINEFEED
920      ISOURCE      JMP Got_time
930      ISOURCE      PBC A,I      ! ELSE PUT CHARACTER INTO
940      ISOURCE      ISZ String      ! STRING AND BUMP COUNT
950      ISOURCE      JMP Read_loop      ! REPEAT
960      ISOURCE Got_time: LDA =18      ! SET UP "PRINTER IS" FOR THE
970      ISOURCE      LDB =80      ! MESSAGE AREA
980      ISOURCE      JSM Printer_select
990      ISOURCE      STA Old_pi      ! SAVE OLD
1000     ISOURCE      STB Old_pw
1010     ISOURCE      LDA =String      ! DO THE PRINT
1020     ISOURCE      JSM Print_string
1030     ISOURCE      JMP Memov      ! JUMP IF MEMORY OVERFLOW
1040     ISOURCE      NOP      ! IGNORE STOP KEY
1050     ISOURCE Restore_pi: LDA Old_pi      ! RESET "PRINTER IS"
1060     ISOURCE      LDB Old_pw
1070     ISOURCE      JSM Printer_select
1080     ISOURCE      RET 1      ! RETURN TO BASIC
1090     ISOURCE Memov: JSM Restore_pi      ! RESTORE THE PRINTER IS
1100     ISOURCE      LDA =2      ! AND GIVE ERROR 2
1110     ISOURCE      JSM Error_exit
1120     ISOURCE !
1130     ISOURCE Isr: LDA =0      ! SIGNAL CARD THAT WE GOT THE
1140     ISOURCE      STA R5      ! INTERRUPT BY DISABLING AND
1150     ISOURCE      LDA =200B      ! THEN RE-ENABLING THE CARD
1160     ISOURCE      STA R5
1170     ISOURCE      LDB Isr_psw      ! TRIGGER EOL BRANCH
1180     ISOURCE      LDA =103B
1190     ISOURCE      STA B,I
1200     ISOURCE      ADB =3
1210     ISOURCE      LDA =Eol_mask
1220     ISOURCE      DIR
1230     ISOURCE      IOR B,I

```

```
1240      ISOURCE      STA B,I
1250      ISOURCE      EIR
1260      ISOURCE      STA Isr_flag,I
1270      ISOURCE      RET 1      ! DONE
1280      ISOURCE      END Time
```


Appendix I

Demonstration Cartridge

Along with the Assembly Language Development and Execution ROMs, a tape cartridge has been provided to demonstrate the capabilities of the assembly language system. This Demonstration Cartridge (HP part number 11141-10154) is specifically intended to —

- Graphically display the kind of speed increases which can be obtained by using assembly language subprograms for certain types of applications.
- Provide a number of programs which can serve as examples of how to write assembly language subprograms.¹
- Provide a set of definitions for some of the special function keys so that those keys can be used as typing aids.

Using the Tape

To run any of the demonstration programs, execute the statement —

```
LOAD "DEMO", 1
```










A set of instructions is displayed which can then be followed interactively.









Typing Aids

The starting and final cursor positions of the typing aids were chosen with assembly listings in mind. The intent in selecting these positions was to make it easy to enter source as it would appear when listed within an assembly listing.

The following table gives, for each key; the typing aid, the position where the typing aid begins, and the position where the cursor will finally reside. Because some typing aids end with a blank, the triangle (Δ) has been chosen to indicate the end of the typing aid. All blanks after the start of the typing aid, and before the triangle, will appear when the key is pressed.

¹ The commented source for the chess program is contained in file CHESSS.

Key	Typing Aid	Typing Aid Starting Position	Final Cursor Position
0	ISOURCE Δ	11	31
1	ISOURCE Δ	11	19
2	ISOURCE ! Δ	11	21
3	 PRINT "A=: "; IMEM(A), "B=: "; IMEM(B) 	home	
4	! Δ	current	current + 2
5	ISOURCE Δ(followed by next line)	11	—
	! Δ	51	53
6	 GET " " Δ	home	6 (over second quote mark in insert character mode)
7	 LOAD " " Δ	home	7 (over second quote mark in insert character mode)
8	 SAVE " " Δ	home	7 (over second quote mark in insert character mode)
9	 STORE " " Δ	home	8 (over second quote mark in insert character mode)
10	 CAT " " Δ	home	6 (over second quote mark in insert character mode)
11	 SCRATCH Δ		
12	(used by other keys)		
13	ISOURCE Δ(followed by next line)	11	—
	: Δ	30	32
14	(Undefined)		
15	(Undefined)		
16	(Undefined)		
17	! Δ	41	43
18	 PRINTER IS Δ	home	12

Key	Typing Aid	Typing Aid Starting Position	Final Cursor Position
19	BIN Δ^1 (use only after using Key 7 or 9)	current - 1	current + 4 (over second quote mark in insert character mode)
20	 RE-SAVE " " Δ	home	10 (over second quote mark in insert character mode)
21	KEY Δ (use only after using key 7 or 9) ²	current - 1	current + 4 (over second quote mark in insert character mode)
22	:F8 Δ	current	current + 3
23	:T15 Δ	current	current + 4
24	(used by other keys)		
25	(used by other keys)		
26	 MASS STORAGE IS " " Δ	home	18 (over second quote mark in insert character mode)
27	 PRINTALL IS Δ	home	13
28	 NORMAL 	home	-
29	 CONT Δ	home	6
30	 PURGE " " Δ	home	8 (over second quote mark in insert character mode)
31	 CREATE " " Δ	home	9 (over second quote mark in insert character mode)

¹ For example, in the insert character mode with the cursor in each case located over the second quote mark:

- Pressing Key 7 results in - LOAD " "
- Pressing Key 19 results in - LOAD BIN " "

² This key performs for the keyword "KEY" as key 19 does for the keyword "BIN". See Note 1.

Appendix J

Error Messages

The following is a numerical list of the system error messages. A brief description of the error is given. For those errors involving the assembly language system, also consult Chapter 9. For all other errors, reference the Operating and Programming Manual.

- | | |
|----|---|
| 1 | Missing ROM; or configuration error |
| 2 | Memory overflow; or subprogram larger than block of memory |
| 3 | Line not found or not in current program segment |
| 4 | Improper return |
| 5 | Abnormal program termination; no END or STOP statement |
| 6 | Improper FOR/NEXT matching |
| 7 | Undefined function or subroutine |
| 8 | Improper parameter matching |
| 9 | Improper number of parameters |
| 10 | String value required |
| 11 | Numeric value required |
| 12 | Attempt to redeclare variable |
| 13 | Array dimensions not specified |
| 14 | Multiple OPTION BASE statement or OPTION BASE statement preceded by variable declarative statements |
| 15 | Invalid bounds on array dimension or string length in memory allocation statement |
| 16 | Dimensions are improper or inconsistent; or more than 32 767 elements in an array |
| 17 | Subscript out of range |

18	Substring out of range; or, string too long
19	Improper value
20	Integer precision overflow
21	Short precision overflow
22	Real precision overflow
23	Intermediate result overflow
24	TAN ($\pi*3/2$), when n is odd
25	Magnitude of argument of ASN or ACS is greater than 1
26	Zero raised to negative power
27	Negative base raised to non-integer power
28	LOG or LGT of negative number
29	LOG or LGT of zero
30	SQR of negative number
31	Division by zero; or, X MOD Y with Y = 0
32	String does not represent valid number; or string response when numeric data required
33	Improper argument for NUM, CHR\$, or RPT\$ function
34	Referenced line is not IMAGE statement
35	Improper format string
36	Out of DATA
37	EDIT string longer than 160 characters
38	I/O function not allowed
39	Function subprogram not allowed
40	Improper replace, delete, or REN command
41	First line number greater than second
42	Attempt to replace or delete a busy line or subprogram
43	Matrix not square

44	Illegal operand in matrix transpose or matrix multiply
45	Nested keyboard entry statements
46	No binary in memory for STORE BIN; or no program in memory for SAVE
47	Subprogram COM declaration is not consistent with main program
48	Recursion in single-line function
49	Line specified in ON declaration not found
50	File number less than 1 or greater than 10
51	File not currently assigned
52	Improper mass storage unit specifier
53	Improper file name
54	Duplicate file name
55	Directory overflow
56	File name is undefined
57	Mass Storage ROM is missing
58	Improper file type
59	Physical or logical end-of-file found
60	Physical or logical end-of-record round in random mode
61	Defined-record size is too small for data item
62	File is protected; or, wrong protect code specified
63	The number of physical records is greater than 32 767
64	Medium overflow (out of user storage space)
65	Incorrect data type
66	Excessive rejected tracks during a mass storage initialization
67	Mass storage parameter less than or equal to 0
68	Invalid line number in GET or LINK operation
69-79	(See Mass Storage ROM errors)

80	Cartridge out; or door open
81	Mass storage device failure
82	Mass storage device not present
83	Mass storage medium is write-protected
84	Record not found
85	Mass storage medium is not initialized
86	Not a compatible tape cartridge
87	Record address error; or, information can't be read
88	Read data error
89	Check read error
90	Mass storage system error
91-99	(See Mass Storage ROM errors)
100	Item in PRINT USING list is string but IMAGE specifier is numeric
101	Item in PRINT USING list is numeric but IMAGE specifier is string
102	Numeric field specifier wider than printer width
103	Item in PRINT USING list has no corresponding IMAGE specifier
104	(See I/O ROM errors)
105-109	(Unused)
110-113	(See Plotter ROM errors)
114-149	(Unused)
150-167	(See I/O ROM errors)
168-183	(Unused)
184-199	(See Assembly Language ROM errors)

System Error {octal number} ; {octal number}

This error indicates an error in the machine's firmware system; it is a fatal error. If reset does not bring control back, the machine must be turned off, then on again. If the problem persists, contact your Sales and Service Office.

Mass Storage ROM Errors

69	Format switch off
70	Not a disk interface
71	Disk interface power off
72	Incorrect controller address; or, controller power off
73	Incorrect device type in mass storage unit specifier
74	Drive missing; or power off
75	Disk system error
76	Incorrect unit code in mass storage unit specifier
77-79	(Unused)
91-99	(Unused)

Plotter ROM Errors

110	Plotter type specification not recognized
111	Plotter has not been specified
112	(Unused)
113	LIMIT specifications out of range
114-119	(Unused)

Assembly Language ROM Errors

184	Improper argument in OCTAL or DECIMAL function
185	Break Table overflow
186	Undefined BASIC label or subprogram name used in IBREAK statement
187	Attempt to write into protected memory; or, attempt to execute instruction not in ICOM region
188	Label used in an assembled location not found
189	Doubly-defined entry point or routine
190	Missing ICOM statement
191	Module not found
192	Errors in assembly
193	Attempt to move or delete module containing an active interrupt service routine
194	Address out of range in IDUMP statement
195	Routine not found
196	Unsatisfied externals
197	Missing COM statement
198	BASIC's common area does not correspond to assembly module requirements
199	Insufficient number of BASIC COM items

Assembly-Time Errors

DD	Doubly-defined label
EN	END instruction missing; or module name does not match
EX	Expression evaluation error
LT	Literal pools full or out of range
MO	ICOM region overflow
RN	Operand out of range
SQ	Argument declaration pseudo-instruction out of sequence
TP	Incorrect type of operand used
UN	Undefined symbol

Appendix **K**

Maintenance

Maintenance Agreements

Service is an important factor when you buy Hewlett-Packard equipment. If you are to get maximum use from your equipment, it must be in good working order. An HP Maintenance Agreement is the best way to keep your equipment in optimum running condition.

Consider these important advantages —

- **Fixed Cost** — The cost is the same regardless of the number of calls, so it is a figure that you can budget.
- **Priority Service** — Your Maintenance Agreement assures that you receive priority treatment, within an agreed-upon response time.
- **On-Site Service** — There is no need to package your equipment and return it to HP. Fast and efficient modular replacement at your location saves you both time and money.
- **A Complete Package** — A single charge covers labor, parts, and transportation.
- **Regular Maintenance** — Periodic visits are included, per factory recommendations, to keep your equipment in optimum operating condition.
- **Individualized Agreements** — Each Maintenance Agreement is tailored to support your equipment configuration and your requirements.

After considering these advantages, we are sure you will see that a Maintenance Agreement is an important and cost-effective investment.

For more information, please contact your local HP Sales and Service Office. A list follows.

HEWLETT PACKARD

SALES & SERVICE OFFICES

AFRICA, ASIA, AUSTRALIA

ANGOLA
Teletra
Empresa Técnica de Equipamentos
Electrónicos, S.A. R.L.
R. Barbosa Rodrigues, 42-102
Caixa Postal, 6487
Luanda
Tel: 35515/6
Cable: TELETRATA Luanda

AUSTRALIA
Hewlett-Packard Australia Pty. Ltd.
31-41 Joseph Street
Blackburn, Victoria 3130
P.O. Box 36
Doncaster East, Victoria 3109
Tel: 896551
Tel: 31-024
Cable: HEWPAUD Melbourne
Hewlett-Packard Australia Pty. Ltd.
31 Bridge Street
Pymble
New South Wales, 2073
Tel: 4496586
Tel: 21561
Cable: HEWPAUD Sydney
Hewlett-Packard Australia Pty. Ltd.
153 Greenhill Road
Parkside, S.A., 5063
Tel: 2725911
Tel: 82636
Cable: HEWPAUD Adelaide
Hewlett-Packard Australia Pty. Ltd.
141 Stirling Highway
Medlands, W.A., 6009
Tel: 385455
Tel: 93859
Cable: HEWPAUD Perth
Hewlett-Packard Australia Pty. Ltd.
121 Wollongong Street
Fyshwick, A.C.T., 2609
Tel: 804244
Tel: 62650
Cable: HEWPAUD Canberra
Hewlett-Packard Australia Pty. Ltd.
5th Floor
Teachers Union Building
495-499 Boundary Street
Spring Hill, Queensland 4000
Tel: 2291544
Cable: HEWPAUD Brisbane

BANGLADESH
The General Electric Co. of Bangladesh Ltd.
Magnus House 72
Dilkusha Commercial Area
Motiluh, Dacca 2
Tel: 252415, 252419
Tel: 734
Cable: GECDAC Dacca

BHUTAN
Hewlett-Packard (Canada) Ltd.
11620A - 168th Street
Edmonton T5M 3T9
Tel: (403) 432-3670
TWX: 610-531-2431

BRITISH COLUMBIA
Hewlett-Packard (Canada) Ltd.
10691 Shalbridge Way
Richmond V2H 2W7
Tel: (604) 270-2277
TWX: 610-825-5059

CHINA
Hewlett-Packard do Brasil
I.E.C. Ltda.
Rua Padre Chagas, 32
90000-Porto Alegre-RS
Tel: (051) 22-2986, 22-5621
Cable: HEWPAUD Porto Alegre
Hewlett-Packard do Brasil
I.E.C. Ltda.
Av. Epitacio Pessoa, 4664
Lagoa
20000-Rio de Janeiro-RJ
Tel: (021) 21905 HPBR-RJ
Cable: HEWPAUD Rio de Janeiro

CHILE
Jorge Calcaín y Cia. Ltda.
Vicuna Mackenna 3, Ofc. 1204
Chileville
Correo 9, Santiago
Tel: 34152
Cable: JCALCAGNI

COLOMBIA
Instrumentación
Henrik A. Langebaek & Kier S.A.
Carrera 7 No. 48-75
Apartado Aéreo 6287
Bogotá, I.D.E.
Tel: 269-8877
Tel: 44400
Cable: AARS Bogotá
Instrumentación
Henrik A. Langebaek & Kier S.A.
Carrera 63 No. 49-A-31
Apartado 54098
Medellín
Tel: 304475

COSTA RICA
Cofinca Costarricense S.A.
Avenida 2, Calle 5
San Pedro de Montes de Oca
Apartado 10159
Zona 9
Tel: 24-38-20, 24-08-19
Tel: 2367 GALGUR CR
Cable: GALGUR

INDONESIA
BERCA Indonesia P.T.
P.O. Box 496Jkt.
Jin. Abdul Muis 62
Jakarta
Tel: 349255, 349886
Tel: 46748 BERSIL IA
Cable: BERSAL
BERCA Indonesia P.T.
P.O. Box 174Gby.
23 Jin. Jimerto
Surabaya
Tel: 42027
Cable: BERCACON
ISRAEL
Electronics Engineering Div.
of Motorola Israel Ltd.
16, Kremenetski Street
P.O. Box 25016
Tel-Aviv
Tel: 36973
Tel: 33669, 34164
Cable: BASTEL Tel-Aviv

JAPAN
Yokogawa-Hewlett-Packard Ltd.
Chuo Bldg., 4th Floor
4-20, Nishinakajima 5-chome
Yokogawa-ku, Osaka-shi
Osaka, 532
Tel: 06-304-6021
Tel: 523-3624
Yokogawa-Hewlett-Packard Ltd.
29-21, Takado-Higashi 3-chome
Suganaka-ku, Tokyo 168
Tel: 03-331-6111
Tel: 232-2024 YHP-Tokyo
Cable: YHPMARKET TOK 23 724

KENYA
Advanced Communications Ltd.
P.O. Box 30070
Nairobi
Tel: 331955
Tel: 22639
Medical Only
International Aeradio (E.A.) Ltd.
P.O. Box 15012
Nairobi Airport
Nairobi
Tel: 336055/56
Tel: 22201/22301
Cable: INTAERIO Nairobi
Medical Only
International Aeradio (E.A.) Ltd.
P.O. Box 95221
Mombasa

KOREA
Samsung Electronics Co., Ltd.
15th Floor, Daeyongak Bldg.
25-5, 1-A
Chong Moo-Ro, Chung-Ku, Seoul
Tel: (21) 6811, 778-3401/2/3/4
Tel: 22575

MACAU
Hewlett-Packard Sales SDN BHD
Suite 2, 21/2, 22
Bangunan Angkasa Raya
Jalan Ampang
Kuala Lumpur
Tel: 23320/27491

INDONESIA
Yokogawa-Hewlett-Packard Ltd.
Nakamo Building
24 Kami Sasajima-cho
Nakamura-ku, Nagoya, 450
Tel: 052 571-5171
Yokogawa-Hewlett-Packard Ltd.
Tanigawa Building
2-24-1 Tsuyasu-cho
Kanagawa-ku, Yokohama, 221
Tel: 045-312-1252
Tel: 382-3204 YHP YOK
Yokogawa-Hewlett-Packard Ltd.
Mito Mitsui Building
105, 1-chome, San-no-maru
Mito, Ibaragi 310
Tel: 0292-25-7470
Yokogawa-Hewlett-Packard Ltd.
Inoue Building
1348-3, Asahi-cho, 1-chome
Atsugi, Kanagawa 243
Tel: 0462-24-0452
Yokogawa-Hewlett-Packard Ltd.
Kumagaya Asahi
Hachijuni Building
4th Floor
3-4, Tsukuba
Kumagaya, Saitama 360
Tel: 0485-24-6563

NEW ZEALAND
Hewlett-Packard (N.Z.) Ltd.
4-12 Cruickshank Street
Kilbirnie, Wellington 3
P.O. Box 9443
Courtenay Place
Wellington
Tel: 877-199
Cable: HEWPAUD Wellington
Hewlett-Packard (N.Z.) Ltd.
Pakuranga Professional Centre
267 Pakuranga Highway
Box 51092
Pakuranga
Tel: 569-651
Cable: HEWPAUD Auckland
Analytical/Medical Only
Medical Supplies N.Z. Ltd.
Scientific Division
79 Carlton Gore Road, Newmarket
P.O. Box 1234
Auckland
Tel: 75-289
Cable: DENTAL Auckland
Analytical/Medical Only
Medical Supplies N.Z. Ltd.
Norrie and Parumona Streets
Porirua
Tel: 75-098
Tel: 3858
Analytical/Medical Only
Medical Supplies N.Z. Ltd.
P.O. Box 309
239 Stanmore Road
Christchurch
Tel: 892-019
Cable: DENTAL Christchurch
Analytical/Medical Only
Medical Supplies N.Z. Ltd.
303 Stratford King Street
Dunedin
Tel: 88-817
Cable: DENTAL Dunedin

PHILIPPINES
The Online Advanced Systems Corporation
Rico House
Amorsolo cor. Herrera Str.
Laguna Village, Makati
P.O. Box 1510
Metro Manila
Tel: 85-35-81, 85-34-91, 85-32-21
Tel: 3274 ONLINE
Rhodesia
Field Technical Sales
45 Kelvin Road North
P.O. Box 3458
Salisbury
Tel: 705231 (5 lines)
Tel: RH 4122

SINGAPORE
Hewlett-Packard Singapore (Pte.) Ltd.
1150 Depot Road
Alexandra P.O. Box 58
Singapore 4
Tel: 270-2355
Cable: HPSPG RS 21486
Cable: HEWPAUD, Singapore

SOUTH AFRICA
Hewlett-Packard South Africa (Pty.) Ltd.
Private Bag Wendywood, Sandton, Transvaal, 2144
Hewlett-Packard Centre
Daphne Street, Wendywood, Sandton
Tel: 802-1040/8
Tel: 88-817
Cable: DENTAL Dunedin

SRI LANKA
Metropolitan Agencies Ltd.
209/9 Union Place
Colombo 2
Tel: 35947
Tel: 1377METROLD CE
Cable: METROLD

SUDAN
Radison Trade
P.O. Box 921
Khartoum
Tel: 44048
Tel: 375

TAIWAN
Hewlett-Packard Far East Ltd.
Taiwan Branch
38 Chung Hsiao West Road
Section 1, 7th Floor
Taipei
Tel: 381960-9, 3141010
Cable: HEWPAUD TAIPEI
Hewlett-Packard Far East Ltd.
Taiwan Branch
68-2, Chung Cheng 3rd. Road
Kaohsiung
Tel: (07) 242318-Kaohsiung
Analytical Only
San Kwang Instruments Co., Ltd.
20 Yung Sui Road
Taipei
Tel: 3615446-9 (4 lines)
Tel: 22894 SANKWANG
Cable: SANKWANG Taipei

TANZANIA
Medical Only
International Aeradio (E.A.) Ltd.
P.O. Box 2577
Kampala
Tel: 54388
Cable: INTAERIO Kampala

ZAMBIA
R.J. Tilbury (Zambia) Ltd.
P.O. Box 2792
Lusaka
Tel: 73793
Cable: ARJAYTEE, Lusaka

CANADA

ALBERTA
Hewlett-Packard (Canada) Ltd.
11620A - 168th Street
Edmonton T5M 3T9
Tel: (403) 432-3670
TWX: 610-531-2431

BRITISH COLUMBIA
Hewlett-Packard (Canada) Ltd.
10691 Shalbridge Way
Richmond V2H 2W7
Tel: (604) 270-2277
TWX: 610-825-5059

MANITOBA
Hewlett-Packard (Canada) Ltd.
380-550 Century St.
Winnipeg R3H 0Y1
Tel: (204) 786-6701
TWX: 610-671-3531

NOVA SCOTIA
Hewlett-Packard (Canada) Ltd.
800 Windmill Road
Dartmouth B3B 1L1
Tel: (902) 469-7820
TWX: 610-671-3531

ONTARIO
Hewlett-Packard (Canada) Ltd.
1020 Morrison Dr.
Ottawa K2H 8K7
Tel: (613) 820-6483
TWX: 610-563-1636
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive
Mississauga L4V 1M8
Tel: (416) 676-9430
TWX: 610-492-4246
Hewlett-Packard (Canada) Ltd.
552 Newbold Street
London N6E 2S5
Tel: (519) 686-9181

QUEBEC
Hewlett-Packard (Canada) Ltd.
275 Hymus Road
Pointe Claire H9R 1G7
Tel: (514) 697-4232
TWX: 610-422-3022
TLX: 05-821-521 HPCL

FOR CANADIAN AREAS NOT LISTED:
Contact Hewlett-Packard (Canada) Ltd. in Mississauga.

CENTRAL AND SOUTH AMERICA

ARGENTINA
Hewlett-Packard Argentina S.A.
Av. Leandro N. Alem 822 - 12°
1001 Buenos Aires
Tel: 31-6063, 4, 5, 6
Tel: 122443 AR CIGY
Cable: HEWPAUDARG
Biotron S.A.C.I. y M.
Bolívar 177
1066 Buenos Aires
Tel: 30-4846, 34-9356, 34-0460,
33-2863
Tel: 011-7595
Cable: Biotron Bares

BOLIVIA
Casa Karin S.A.
Calle Potosí 1130
P.O. Box 500
La Paz
Tel: 41530, 53221
Tel: CWC BX 5298, ITT 3560082
Cable: KAVLIN

BRAZIL
Hewlett-Packard do Brasil
I.E.C. Ltda.
Alameda Rio Negro, 750
Alphaville
06400 Barueri SP
Tel: 429-3222
Cable: HEWPAUD Sao Paulo

CHILE
Jorge Calcaín y Cia. Ltda.
Vicuna Mackenna 3, Ofc. 1204
Chileville
Correo 9, Santiago
Tel: 34152
Cable: JCALCAGNI

COLOMBIA
Instrumentación
Henrik A. Langebaek & Kier S.A.
Carrera 7 No. 48-75
Apartado Aéreo 6287
Bogotá, I.D.E.
Tel: 269-8877
Tel: 44400
Cable: AARS Bogotá
Instrumentación
Henrik A. Langebaek & Kier S.A.
Carrera 63 No. 49-A-31
Apartado 54098
Medellín
Tel: 304475

COSTA RICA
Cofinca Costarricense S.A.
Avenida 2, Calle 5
San Pedro de Montes de Oca
Apartado 10159
Zona 9
Tel: 24-38-20, 24-08-19
Tel: 2367 GALGUR CR
Cable: GALGUR

ECUADOR
Computadoras y Equipos Electrónicos del Ecuador
P.O. Box 6423 CCI
Eloy Alfaro No. 1824, 3° Piso
Quito
Tel: 453 482
Tel: 5428 CYEDE ED
Cable: CYEDE-Quito
Medical Only
Hospital S.A.
Casilla 3590
Robles 625
Quito
Tel: 545-250
Cable: HOSPITALAR-Quito

EL SALVADOR
Medellín
Bulevar de los Heroes 11-48
San Salvador
Tel: 252787

GUATEMALA
IPESA
Avenida Reforma 3-48,
Zona 9
Guatemala City
Tel: 316627, 314786, 66471-5, ext. 9
Tel: 4192 Teletro Gu

MEXICO
Hewlett-Packard Mexicana, S.A. de C.V.
Av. Periferico Sur No. 6501
Teseo, Xochimilco
Mexico 23, D.F.
Tel: 905-676-4600
Tel: 017-74-507
Hewlett-Packard Mexicana, S.A. de C.V.
Ave. Constitución No. 2184
Monterrey, N.L.
Tel: 48-71-32, 48-71-84
Tel: 038-410
NICARAGUA
Roberto Terán G.
Apartado Postal 689
Edificio Terán
Managua
Tel: 25114, 23412, 23454, 22400
Cable: ROTERAN Managua

PANAMA
Electrónico Balboa, S.A.
Apartado 4929
Panama 5
Calle Samuel Lewis
Cuidad de Panama
Tel: 64-2700
Tel: 3483103 Curundu,
Canal Zone
Cable: ELECTRON Panama

PERU
Compañía Electro Médica S.A.
Los Flamencos 145
San Isidro Casilla 1030
Lima 1
Tel: 41-4325
Tel: Pub. Booth 25424 SISIDRO
Cable: ELMED Lima

SURINAME
Surter Radio Holland N.V.
Grote Hofstr. 3-5
P.O. Box 155
Paramaribo
Tel: 72118, 77880
Cable: Surter

TRINIDAD & TOBAGO
CARTEL
Caribbean Telecoms Ltd.
P.O. Box 732
69 Frederick Street
Port-of-Spain
Tel: 62-53068

URUGUAY
Pablo Ferrando S.A.C. el.
Avenida Italia 2877
Casilla de Correo 370
Montevideo
Tel: 40-3102
Tel: 702 Public Booth Para
Pablo Ferrando
Cable: RADIUM Montevideo

VENEZUELA
Hewlett-Packard de Venezuela C.A.
P.O. Box 50933
Caracas 105
Los Ruices Norte
3a Transversal
Edificio Segre
Caracas 107
Tel: 239-4133 (20 lines)
Tel: 25146 HEWPAUD
Cable: HEWPAUD Caracas

FOR AREAS NOT LISTED, CONTACT:
Hewlett-Packard Inter-Americas
3200 Hillview Ave.
Palo Alto, California 94304
Tel: (415) 856-1501
TWX: 910-373-1260
Cable: HEWPAUD Palo Alto
Tel: 034-8300, 034-8493

EUROPE, NORTH AFRICA AND MIDDLE EAST

AUSTRIA
Hewlett-Packard Ges.m.b.H.
Händelskai 52
P.O. Box 7
A-1205 Vienna
Tel: 351621-27
Cable: HEWPAK Vienna
Telex: 75923 hewpak a

BAHRAIN
Medical Only
Wael Pharmacy
P.O. Box 548
Bahrain
Tel: 54886, 56123
Telex: 8550 WAEI GJ
Cable: WAEI/PHARM

Bahrein
Analytical Only
Al Hamdy Trading
and Contracting
P.O. Box 20074
Manama
Tel: 259978, 259958
Telex: 8985 KALDIA GJ

BEELGIUM
Hewlett-Packard Benelux
S.A.N.V.
Avenue du Col-Vert, 1,
(Groenkraklaan)
B-1170 Brussels
Tel: (02) 680 50 50
Cable: PALOBEN Brussels
Telex: 23-494 paloben br

CYPRUS
Kyprionics
19 Gregorios Xenopoulos Street
P.O. Box 1152
Nicosia
Tel: 45628/29
Cable: Kyprionics Pandeis
Telex: 3018

CZECHOSLOVAKIA
Vývojova a Provozní Zakladna
Výzkumných Ústavů v Brněch
ČSSR-25087 Brno
Tel: 89 93 41
Telex: 12133

DDR
Entwicklungslabor der TU Dresden
Forschungsinstitut Meinsberg
DDR-7305
Waldheim/Meinsberg
Tel: 37 667
Telex: 518741

Export Contact AG Zurich
Guenther Forbiger
Schlegelstrasse 15
1040 Berlin
Tel: 42-74-12
Telex: 111889

DENMARK
Hewlett-Packard A/S
Ditlevsvej
DK-3480 Birkerød
Tel: (02) 81 66 40
Cable: HEWPAK AS
Telex: 37409 hps dk

Hewlett-Packard A/S
Navegvej 1
DK-8500 Silkeborg
Tel: (06) 82 71 66
Cable: HEWPAK AS
Telex: HEWPAK AS

EGYPT
I.E.A.
International Engineering Associates
24 Hussein Hegazi Street
Kasr-el-Aini
Cairo
Tel: 23 829
Telex: 30830
Cable: INTENGASSO

SAMITRO
Sami Amin Trading Office
18 Abdel Aziz Gawish
Abdine-Cairo
Tel: 24932
Cable: SAMITRO CAIRO

FINLAND
Hewlett-Packard Oy
Nahkahuusuntie 5
P.O. Box 6
SF-00211 Helsinki 21
Tel: (90) 6923031

FRANCE
Hewlett-Packard France
Avenue des Tropiques
Les Ulis
Boite Postale No. 6
91041 Orsay-Cedex
Tel: (1) 907 78 25
TWX: 600048F

Hewlett-Packard France
Chemin des Moulins
8 P. 162
91300 Evry
Tel: (78) 33 61 25,
TWX: 310617F

Hewlett-Packard France
Périmètre de la Cépère
31081 Toulouse-Le Mirail
Tel: (05) 140 11 12
TWX: 510957F

Hewlett-Packard France
Le Lignou
Bureau de vente de Marseille
Rue de la République
13100 Aix-en-Provence
Tel: (42) 59 41 02
Hewlett-Packard France
2, Allée de la Bourgnette
35100 Rennes
Tel: (09) 51 42 44
TWX: 740912F

Hewlett-Packard France
16, rue du Canal de la Marne
67300 Schiltigheim
Tel: (88) 83 08 10
TWX: 890141F

Hewlett-Packard France
Immeuble périmètre
Rue van Gogh
59650 Villeneuve D Ascq
CCSP-25087 Brno
Tel: (20) 91 41 25
TWX: 150124F

Hewlett-Packard France
Bureau de Vente
Centre d'affaires Paris-Nord
Bâtiment Ampère
Rue de la Commune de Paris
B.P. 300
93153 Le Blanc Mesnil Cedex
Tel: (01) 931 88 50

Hewlett-Packard France
Av. du Pdt. Kennedy
35700 Montgerais
Tel: (56) 97 22 69
Hewlett-Packard France
"France-Evy" Immeuble Lorraine
Boilevard de France
91035 Evry-Cedex
Tel: 077 96 60

Hewlett-Packard France
91035 Evry-Cedex
Tel: 077 96 60

Hewlett-Packard France
60, Rue de Metz
57130 Jouy aux Arches
Tel: (87) 69 45 32

GERMAN FEDERAL REPUBLIC
Hewlett-Packard GmbH
Vertriebszentrale Frankfurt
Bernar Strasse 117
Postfach 580 140
D-6000 Frankfurt 56
Tel: (0611) 50-04-1
Cable: HEWPAKAS Frankfurt
Telex: 04 13249 hpfm d

Hewlett-Packard GmbH
Technisches Büro Böblingen
Herrnberger Strasse 110
D-7030 Böblingen, Württemberg
Tel: (0703) 867-1
Cable: HEWPAKAS Böblingen
Telex: 0726573 bbn

Hewlett-Packard GmbH
Technisches Büro Düsseldorf
Emanuel-Leutze-Str. 1 (Seestern)
40000 Düsseldorf
Tel: (0211) 59711
Telex: 08586 533 hppd d

Hewlett-Packard GmbH
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

Hewlett-Packard GmbH
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

Hewlett-Packard GmbH
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

Hewlett-Packard GmbH
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

Cable: HEWPAKAS Hamburg
Telex: 21 63 032 hphd d
Hewlett-Packard GmbH
Technisches Büro Hannover
Am Grossmarkt 6
D-3000 Hannover 91
Tel: (0511) 46 60 01
Telex: 092 3259

Hewlett-Packard GmbH
Technisches Büro Nürnberg
Neumeyerstrasse 90
D-85000 Nürnberg
Tel: (0911) 56 30 83
Telex: 0623 806

Hewlett-Packard GmbH
Technisches Büro München
Eichenstrasse 5
D-8021 Taufkirchen
Tel: (089) 6117-1

Hewlett-Packard GmbH
Technisches Büro Berlin
Katharinenstrasse 4
D-1000 Berlin 30
Tel: (030) 24 00 86
Telex: 018 3405 hpbld d

GREECE
Kostas Karayannis
8 Omirou Street
Athens 133
Tel: 32 30 303/82/37 731
Analytical Only
INTECO
G. Papanastasiou & Co.
17 Marm Street
Athens 103
Tel: 5522 915/5221 989
Telex: 21 5329 INTE GR

HUNGARY
MTA
Műszertudományi és Mérési Technikai
Szolgálat
Lenin Krt. 67, P.O. Box 241
1391 Budapest VI
Tel: 42 03 38
Telex: 22 51 14

ICELAND
Medical Only
Elding Trading Company Inc.
Hafnarhvoli - Tryggvagötu
P.O. Box 895
Tel: 851082-5
Cable: ELDING Reykjavik
Telex: 213405 hewp ir

IRAN
Hewlett-Packard Iran Ltd.
No. 13, Fourteenth St.
Tehran
Tel: 851082-5
Cable: ELDING Reykjavik
Telex: 213405 hewp ir

IRELAND
Hewlett-Packard Ltd.
King Street Lane
Winnesh, Wokingham
Berk, RG11 5AR
GB-England
Tel: (0734) 78 47 74
Cable: HEWPAK Ltd
Telex: 847178

Hewlett-Packard Ltd.
Technisches Büro Böblingen
Herrnberger Strasse 110
D-7030 Böblingen, Württemberg
Tel: (0703) 867-1
Cable: HEWPAKAS Böblingen
Telex: 0726573 bbn

Hewlett-Packard Ltd.
Technisches Büro Düsseldorf
Emanuel-Leutze-Str. 1 (Seestern)
40000 Düsseldorf
Tel: (0211) 59711
Telex: 08586 533 hppd d

Hewlett-Packard Ltd.
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

Hewlett-Packard Ltd.
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

Hewlett-Packard Ltd.
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

Hewlett-Packard Ltd.
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

Hewlett-Packard Ltd.
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

Hewlett-Packard Ltd.
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

Hewlett-Packard Ltd.
Technisches Büro Hamburg
Wendenstrasse 23
D-2000 Hamburg 1
Tel: (053) 173-2600

ITALY
Hewlett-Packard Italiana S.p.A.
Via G. Di Vittorio, 9
20063 Cernusco
Sul Naviglio (MI)
Tel: (2) 905691
Telex: 311046 HEWPAKIT

Hewlett-Packard Italiana S.p.A.
Via Turazza, 14
35100 Padova
Tel: (49) 664888
Telex: 41612 HEWPAKIT

Hewlett-Packard Italiana S.p.A.
Via G. Armetlini 10
1-00143 Roma
Tel: (06) 54 69 61
Telex: 61514

Hewlett-Packard Italiana S.p.A.
Corso Giovanni Lanza 94
I-10133 Torino
Tel: (011) 882245/659308
Medical/Calculators Only
Hewlett-Packard Italiana S.p.A.
Via Principe Nicola 43 G.C.
I-95126 Catania
Tel: (095) 37 05 04

Hewlett-Packard Italiana S.p.A.
Via Nuova San Rocco A.
I-80131 Napoli
Tel: (081) 7913544
Hewlett-Packard Italiana S.p.A.
Via E. Masi, 9/6
I-40137 Bologna
Tel: (051) 307887/900040

JORDAN
Moussier Cousins Co.
P.O. Box 1387
Amman
Tel: 24907/3907
Telex: SABCO JO 1456
Cable: MOUASHERCO

KUWAIT
Al-Khalifa Trading &
Contracting
P.O. Box 830-Salat
Kuwait
Tel: 42 4910/41 1726

LUXEMBURG
Hewlett-Packard Benelux
S.A. A.V.
Avenue du Col-Vert, 1
(Groenkraklaan)
B-1170 Brussels
Tel: (02) 672 22 40
Cable: PALOBEN Brussels
Telex: 23 494

MOROCCO
Dolbeur
81 rue Karachi
Casablanca
Tel: 3041 82
Telex: 2205122822
Cable: MATERIO
Gerep
3, rue d'Agadir
Casablanca
Tel: 272093-6
Telex: 23739
Cable: GEREP-CASA

NETHERLANDS
Hewlett-Packard Benelux N.V.
Van Heuven Goedhardstraat 121
P.O. Box 667
NL-Amstelveen 1134
Tel: (020) 47 20 21

NORWAY
Hewlett-Packard Norge AS
Osteren Miersiers
P.O. Box 34
1345 Osteras
Tel: (021) 1711 80
Telex: 15621 hpnas n

Hewlett-Packard Norge AS
Nymadsvej 114
5000 Bergen

POLAND
Biuro Informacji Technicznej
Hewlett-Packard
Ul. Stawki 2, 6P
00-950 Warszawa
Tel: 33 25 88/39 67 43
E-811ba0

UNIPAN
Biuro Obsługi Technicznej
01-447 Warszawa
ul. Nowelska 6
Poland
Zakłady Naprawcze Sprzetu
Medycznego
Plac Komuny Paryskiej 6
90-007 Lodz
Tel: 334-41, 337-83
Telex: 886981

PORTUGAL
Teletra-Empresa Técnica de
Equipamentos Eléctricos S.A. L.
Rua Rodrigo da Fonseca 103
P.O. Box 2531
P-Lisbon 1
Tel: (1) 98 60 72
Cable: TELETRA Lisbon
Telex: 12598

Medical only
Mundinter
Intercambio Mundial de Comercio
S. A.
P.O. Box 2761
Avenida Antonio Augusto
de Aguiar 138
P - Lisbon
Tel: (19) 53 21 31/7
Telex: 16991 munter p

Cable: INTERCAMBIO Lisbon

QATAR
Nasser Trading & Contracting
P.O. Box 1563
Doha
Tel: 22170
Telex: 4439 NASSER
Cable: NASSER

ROMANIA
Hewlett-Packard Reprezentanta
Bd. n. Baicescu 16
Bucuresti
Tel: 15 80 21/33 88 85
Telex: 10440
I.R.U.C.
Intreprinderea Pentru
Infritinerea
SI Repararea Utilajelor de Calcul
B-dul Prof. Dimitrie Pompei 6
Bucuresti-Sectorul 2
Tel: 88-20-70, 88-24-40, 88-67-95
Telex: 118

SAUDI ARABIA
Modern Electronic
Establishment (Head Office)
P.O. Box 1228, Baghdadiah Street
Jeddah
Tel: 27 798
Telex: 40035
Cable: ELECTA JEDDAH

Modern Electronic
Establishment (Branch)
P.O. Box 2728
Riyadh
Tel: 62596/66232
Cable: RAOUFCO

Modern Electronic
Establishment (Branch)
P.O. Box 193
Al-Khobar
Tel: 44678-44813

SPAIN
Hewlett-Packard Española, S.A.
Calle Jerez, 3
E-Madrid 16
Tel: (1) 456 26 00 (10 lines)
Telex: 32515 hpe

Hewlett-Packard Española S.A.
Colonia Mirasierra
Edificio Juban
% Costa Brava, 13
Madrid 34
Tel: (1) 456 26 00 (10 lines)
Telex: 32515 hpe

Hewlett-Packard Española S.A.
Milesad 21-23
NORTH CAROLINA
Tel: (3) 203 6200 (5 lines)

NEW YORK
6 Automation Lane
Computer Park
Tulsa 77025
Tel: (503) 458-1550
TWX: 710-444-4961

650 Perinton Hill Office Park
Fairport 14450
Tel: (516) 223-9950
TWX: 516-253-0092

No. 1 Pennsylvania Plaza
55th floor
34th street & 8th Avenue
New York 10001
Tel: (212) 971-0800

5585 East Molloy Road
Syracuse 13211
Tel: (315) 455-2486
1 Crossways Park West
Woodbury 11797
Tel: (516) 821-0300

5605 Roanne Way
Greensboro 27405
Tel: (919) 852-1800

Hewlett-Packard Española, S.A.
Av. Ramón y Cajal, 1
Edificio Sevilla, planta 9
Seville 5
Tel: 64 44 54/58

Hewlett-Packard Española S.A.
Edificio Albia II 7º B
E-811ba0
Tel: 23 83 06/23 82 06
Hewlett-Packard Española S.A.
C/Ramon Gordillo 1
Enllo
E-Valladolid-10
Tel: 96-351.13.54/061.13.58

SWEDEN
Hewlett-Packard Sverige AB
Enighetsvägen 3, Fack
S-161 Bromma 20
Tel: (08) 730 05 50
Berk. RG11 5AR
Cable: MEASUREMENTS
Stockholm

Hewlett-Packard Sverige AB
S-421 32 Västra Frölunda
Tel: (031) 49 09 50
Telex: 10721 via Bromma office

SWITZERLAND
Hewlett-Packard (Schweiz) AG
Zürcherstrasse 20
S. A.
P.O. Box 307
CH-8952 Schlieren-Zürich
Tel: (011) 7305240
Telex: 53833 hpag ch
Cable: HPAG CH

Hewlett-Packard (Schweiz) AG
Château Bloc 19
CH-1219 Le Lignon-Geneve
Tel: (022) 96 03 22
Telex: 57233 hpag ch
Cable: HEWPAKAG Geneva

SYRIA
General Electronic Inc.
Nuri Basha-Ahmad Ebn Kays Street
P.O. Box 5781
Damascus
Tel: 33 24 87
Telex: 11215 IITKAL
Cable: ELECTROBOR DAMASCUS

Medical/Personal calculator only
Sawah & Co.
Place Amé
B.P. 2308
Damascus
Tel: 16 367-19 697-14 268
Suleiman Hilal El Mawi
P.O. Box 232
Mamoun Bita Street, 56-58
Damascus Tel: 11 46 63

TUNISIA
Tunisie Electronique
11 Avenue de la Liberté
Tunis
Tel: 280 144
Corema
1 ter. Av. de Carthage
Tunis
Tel: 253 821
Telex: 12319 CABAM TN

TURKEY
TEKIN Company Ltd.
Riza Sah Pehlivi
Caddesi No. 7
Kavaddere, Ankara
Tel: 42155
Tekin Com., Ltd.
Barbaros Bulvarı 55/12
Besikyas, Istanbul
Tel: 513 546
Telex: 23540

Medical only
E.M.A.
Muhendisk Kollektiv Sirketi
Mehmet Eadem Sokak 41/6
Kizilca Caddesi
Ankara
Tel: 17 56 22
Cable: EMATRADE/Ankara

Medical only
Yilmaz Özyürek
Mili Müdairesi Cad 16/6
Kizilay

OREGON
17890 SW Lower Boones
Ferry road
Tulstien 97062
Tel: (503) 620-3350
TWX: 710-444-4961

650 Perinton Hill Office Park
Fairport 14450
Tel: (516) 223-9950
TWX: 516-253-0092

No. 1 Pennsylvania Plaza
55th floor
34th street & 8th Avenue
New York 10001
Tel: (212) 971-0800

5585 East Molloy Road
Syracuse 13211
Tel: (315) 455-2486
1 Crossways Park West
Woodbury 11797
Tel: (516) 821-0300

5605 Roanne Way
Greensboro 27405
Tel: (919) 852-1800

ANKARA
Tel: 25 03 09 - 17 80 26
Telex: 42576 ÖZEK TR
Cable: ÖZYUREK ANKARA

UNITED ARAB EMIRATES
Emirat Ltd. (Head Office)
P.O. Box 1641
Sharjah
Tel: 354121/3
Telex: 8136
Emirat Ltd. (Branch Office)
P.O. Box 2711
Anu Dhabi
Tel: 331370/1

UNITED KINGDOM
Hewlett-Packard Ltd.
King Street Lane
Winnesh, Wokingham
Berk. RG11 5AR
Tel: (0734) 784774
Telex: 847178/9

Hewlett-Packard Ltd.
Trafalgar House
Navigation Road
Aldershot
Hants GU11 1NU
Tel: (0161) 928 8422
Telex: 668068

Hewlett-Packard Ltd.
Lyon Court
Hertfordshire
Dunstable Road
Halesowen
West Midlands B62 8SD
Tel: (021) 550 9911
Telex: 339105

Hewlett-Packard Ltd.
Wedgwood House
799, London Road
Thornton Heath
Surrey CR4 6XL
Tel: (011) 6840103
Telex: 946825

Hewlett-Packard Ltd
10, Wesley St.
Castletown
York WF10 1AE
Tel: (01977) 550616
Telex: 557535

Hewlett-Packard Ltd
1, Wallace Way
Hitchin
Hertfordshire SG4 0SE
Tel: (0462) 31111
Telex: 82 59 81

USSR
Hewlett-Packard
Representative Office USSR
Pokrovsky Boulevard 4/17-kv 12
Moscow 101000
Tel: 294 20 24
Telex: 7825 hewpak su

YUGOSLAVIA
Iskra-Standard Hewlett-Packard
Miklosiceva 38/VII
61000 Ljubljana
Tel: 31 58 79/32 16 74

SOCIALIST COUNTRIES
NOT SHOWN PLEASE
CONTACT:
Hewlett-Packard Ges.m.b.H.
Händelskai 52
P.O. Box 7
A-1205 Vienna, Austria
Tel: (0222) 35 16 21 to 27

**MEDITERRANEAN AND
MIDDLE EAST COUNTRIES**
NOT SHOWN PLEASE
CONTACT:
Hewlett-Packard S.A.
Mediterranean and Middle
East Operations
35, Kolokotroni Street
Platia Kefallionu
GR-Kissia, Athens, Greece
Tel: 8080337/259429

FOR OTHER AREAS
NOT LISTED CONTACT
Hewlett-Packard S.A.
7, rue du Bois-du-Lan
P.O. Box
CH-1217 Meyrin 2 - Geneva
Switzerland
Tel: (022) 82 70 00

UNITED STATES

ALABAMA
P.O. Box 4207
8290 Whitesburg Dr.
Huntsville 35802
Tel: (205) 881-4591

8933 E. Roebuck Blvd.
Birmingham 35206
Tel: (205) 836-2203/2

ARIZONA
2338 E. Magnolia St.
Phoenix 85034
Tel: (602) 244-1361
2424 East Aragon Rd.
Tucson 85706
Tel: (602) 689-4661

***ARKANSAS**
Medical Service Only
P.O. Box 5646
Brady Station
Little Rock, 72215
Tel: (501) 376-1844

Subject Index

a

AAR 40,207,221,222
 ABR 40,207,221,222
 ABS function 101
 ACS function 101,266
 ADA 35,207,221,222
 ADB 35,207,221,222
 AND:
 instruction 41,207,221,222
 operation 101
 ANY 112,128,128,196,224
 ASC declaration 182
 ASCII character set 199
 ASMB file-type 19,22,23
 ASN function 101,266
 ASSIGN 163,164,165
 ATN function 101
 AUTO 11,14
 Abortive access 142
 Absolute 75
 Access:
 abortive 142
 asynchronous 142
 granting 142
 synchronous 142
 Accumulators:
 General 26,34,35,40,69,231
 map 27
 Addition, General 35
 Addition,BCD 210
 Addition,binary 207
 Address, machine 78
 Addressing:
 General 33
 indirect 34,146
 Arguments:
 changing values of 122
 passing from BASIC 109
 system information about 113
 Arrays:
 changing values in 124
 identifiers 129
 obtaining information on 115
 retrieving elements from 118
 retrieving substrings from 121
 system information about 113

Assembled location 225
 Assembled location,definition 4
 Assembling:
 process 60
 error 194
 Assembly:
 Development ROM 261
 Execution ROM 261
 Language ROMs 268
 conditional, definition 4
 Asynchronous access 142

b

BASIC:
 General 8,9,11,12,122,174
 assembly language extensions 13
 assembly source entry 14
 branching on interrupts 150
 calling assembly language 7
 common 128,132
 comparison of expressions 77
 comparison of operators 76
 drivers 10
 end-of-line branches 139
 labels 50,52
 passing variables 5
 relation to assembly language 49
 role of STEP key 172
 routines 175
 subprograms 155,163,176,179
 variables:
 general 123,175
 names 17
 structure 30
 BCD:
 General 31
 Math group 44,45
 addition 210
 arithmetic:
 General 83
 addition 85
 subtraction 86
 defined 83
 division 46,92,210
 multiplication 46,90,210
 normalization 44,89,213
 registers 84
 rounding 89

BIN declaration 182
 BSS 56,57,75,224
 Backplane 133,134
 Base page 29,33,221
 Base page,defined 30
 Binary Processor Chip (BPC) 25,26
 Blank lines, in listings 65
 Blind parameters 112
 Booth's algorithm 35,212
 Braces (in syntax), explained 6
 Brackets (in syntax), explained 6
 Branch group 36
 Branching:
 General 36,211
 end-of-line 139
 interrupt,prioritizing 153
 on interrupts 150
 Break points 172,270
 Buffers,device 157,158,162
 Bus cycles, I/O 25,33,187,207,208
 210,213 – 216,218,219,220
 Bus, I/O 47
 Busy bits 130
 Busy bits, definition 4
 Busy utility 81,130
 Bytes:
 General 42,43
 definition 4
 pointers 69
 retrieving from BASIC 119

c

CALL 19,111
 CBL 43,208,221,222
 CBU 43,208,221,222
 CDC 46,85,208,221,222
 CHR\$ 266
 CLA 41,208,221,222
 CLB 41,208,221,222
 CLR 34,146,208,221,222
 CMA 41,208,221,222
 CMB 41,208,221,222
 CMX 45,86,208,221,222
 CMY 46,86,208,221,222
 COM:
 pseudo-instruction 59,128,129,195
 196,224
 statement 115,128,195,267,270
 CONTINUE key 111,171,175
 COS function 101

CPA 37,208,221,222
 CPB 37,208,221,222
 Clock times 221
 Code:
 object 5,7,8,9,14,60,68
 source 5,60,189
 Commands:
 AUTO 11,14
 EDIT 11,11,14,14,53
 REN 266
 SCRATCH A 21,143,159,162,194
 SCRATCH C 21,193
 Comments,in assembly source 51,53,54
 Common 128,132
 Common, error regarding 195
 Complement:
 one's 41,208
 ten's 45,86,208
 two's 46,219
 Conditional assembly:
 definition 4
 flags 67
 general 60,65
 Control registers 135
 Current page:
 General 29,33,221
 defined 30

d

DAT 57,224
 DATA 266
 DBL 43,209,221,222
 DBU 43,209,221,222
 DDR 47,149,209,221,222
 DEC declaration 182
 DECIMAL 16,101,169,184,193,225,270
 DIR 47,156,209,221,222
 DISABLE 156
 DIV function 101
 DMA instruction 149,209,221,222
 DROUND 101
 DRS 45,209,221,222
 DSZ 38,209,221,222
 Data generators 57
 Data locations 177
 Debugging 2,8,169
 Decimal Carry flag 46,85,89,180,186,208
 209,211,213,216

- Declarations:
- ANY 112,128,196
 - ASC 182
 - BIN 182
 - DEC 182
 - FIL 110,128,196
 - HEX 182
 - INT 110,128,196
 - OCT 182
 - REL 110,128
 - SHO 110,128,196
 - STR 110,111,128,196
- Device buffers 157,158,162
- Direct memory access (DMA):
- General .. 47,47,133,141,142,147,216
 - lockout time 221
 - registers 69,148,231
 - timings 221
 - transfers 149
- Division, BCD 46,92,210
- Dot matrix, explained 6
- Dumps 181
- files 267
- mass storage 269,269
- messages:
- General 265
 - assembly-time 193,271
 - run-time 193
 - processing 189
 - run-time 190
 - syntax-time 189
- Exclamation point 53
- Expressions:
- General 75,196
 - absolute, defined 75
 - external 196
 - octal, defined 5
 - relocatable 196
 - relocatable, defined 75
 - type of result 76
- Extend flag 35,38,39,145,180,186,207
216,217,219
- Extended Math Chip (EMC) 25
- External 76,196

e

- EDIT 11,14,53,266
- EIR 47,156,209,221,222
- EJECT option, IASSEMBLE statement
..... 60,63
- ENABLE 156
- END:
- pseudo-instruction ... 5,11,17,18,55,59
195,224,271
 - statement 195,265
- ENT 77,194,224
- EQU 4,59,71,75,197,224,225
- ERRL 101,192
- ERRM\$ 192
- ERRN 101,192
- EXE 47,196,210,221,222
- EXOR 101
- EXP function 101
- EXT 59,77,78,80,195,197,224
- Ellipses (in syntax), explained 6
- Entry points 77,194
- Error_exit utility 81,186,191
- Errors:
- assembly language 270,270
 - assembly-time 190

f

- FDV 46,94,210,221,222
- FIL 110,128,196,224
- FMP 46,90,91,92,207,221,222
- FOR statement 268
- FRACT 101
- FXA 46,87,89,90,210,221,222
- Files:
- ASMB-type 19,22,23
 - OPRM-type 19,22,23
 - descriptor 163,164,165
 - errors 267
 - names 17
 - numbers 117,118
- Flag line 47,217
- Flags:
- Conditional assembly 67
 - Decimal Carry .. 46,85,89,180,186,208
209,210,211,213,216
 - Extend . 35,38,39,39,145,180,186,205
214,217,217
 - Overflow .. 35,38,39,145,180,186,205
215,218,217
- Full-precision numbers ... 30,58,78,84,102
103,104,105,117,118

Functions:

ABS	101
ACS	101,266
ASN	101,266
ATN	101
CHR\$	266
COS	101
DECIMAL	16,101,169,184,193 225,270
DIV	101
DROUND	101
ERRL	101,192
ERRM\$	192
ERRN	101,192
EXP	101
FRACT	101
IADR	16,101,169,184,185,226
IMEM	16,101,169,184,186
INT	101
LGT	101,266
LOG	101,266
NUM	266
OCTAL	16,101,169,184,193,229,270
PI	101
PROUND	101
RES	101
RND	101
RPT\$	266
SGN	101
SIN	101
SQR	101,266
TAN	101,266
TYP	101

g

GET statement	53,177,268
Get_bytes utility	81,116,119,120,128
Get_elem_bytes utility	81,116,120,128
Get_element utility	81,116,118,121,128
Get_file_info utility	81,164
Get_info utility	81,114,118,121,123,128
Get_value utility	81,116–119,128
Groups:	
BCD Math	44,45
Branch	36
I/O	47
Integer Math	35
Logical	41
Stack	42,146
Test/Alter/Branch	38
Test/Branch	37

h

HED	64,65,224
HEX declaration	182

i

I/O:

ROM	268
bus	47
bus cycles	25,33,187
group	47
interrupt	133,138
operations,relation to busy bits	130
programmed	133,138
registers	26,26,70,187,231
sample programs	237
IADR	16,101,169,184,185,226
IASSEMBLE	12,16,20,60,62,65,190 194,226
IASSEMBLE ALL	60
IBREAK	16,169,174,179,181,186,193 227,270
IBREAK ALL	169,178,227
IBREAK DATA	169,177,179,180,193,227
ICALL	12,16,18,19,107,107,108,115 117,123,131,172,227
ICHANGE	16,169,187,227
ICOM	12,16,18,19,21,22,23,193,196 227,270,271
ICOM region	19–23,28,56,108,157,161 169,178,193,194,196,232,270,271
IDELETE	16,18,22,227
IDELETE ALL	227
IDUMP	16,169,181,186,194,228,270
IF conditional	66,67
IFA	66,224
IFB	66,224
IFC	66,224
IFD	66,224
IFE	66,224
IFF	66
IFG	66,224
IFH	66,224
IFP	66,224
ILOAD	14,16,18,20,22,55,171,194,228
IMAGE	268
IMEM	16,101,169,184,186,228
INORMAL	16,169,179,193,228

- INT:
 - function 101
 - pseudo-instruction ... 110,128,196,224
- IOR 41,210,221,222
- IPAUSE OFF 16,169,174,228
- IPAUSE ON 16,169,171,175,229
- ISOURCE 11,11,49,50,53,54,63,229
- ISTORE 14,16,19,23,55,194,229
- ISTORE ALL 24
- ISZ 38,211,221,222
- Index 277
- Indirect addressing:
 - General 34,68
 - in ISRs 146
- Input cycle, explained 25
- Input-Output Controller (IOC) 25,26
- Instructions:
 - individual execution of 170
 - machine:
 - General 32,54,223
 - AAR 40,207
 - ABR 40,207
 - ADA 35,207
 - ADB 35,207
 - AND 41,208
 - CBL 43,208
 - CBU 43,208
 - CDC 46,85,208
 - CLA 41,208
 - CLB 41,208
 - CLR 34,146,208
 - CMA 41,208
 - CMB 41,208
 - CMX 45,86,208
 - CMY 46,86,208
 - CPA 37,208
 - CPB 37,209
 - DBL 43,209
 - DBU 43,209
 - DDR 47,149,209
 - DIR 47,156,209
 - DMA 47,149,209
 - DRS 45,209
 - DSZ 38,209
 - EIR 47,156,209
 - EQU 4,225
 - EXE 47,210
 - FDV 46,94,210
 - FMP 46,90,91,92,210
 - FXA 46,87,89,90,210
 - IOR 41,210
 - ISZ 38,211
 - JMP 36,211
 - JSM 36,79,211,214
 - LDA 34,211,213
 - LDB 34,211
 - MLY 45,211
 - MPY 35,212
 - MRX 44,87,89,91,212
 - MRY 45,87,89,91,92,212
 - MWA 46,90,213
 - NOP 47,213
 - NRM 45,89,213
 - PBC 43,213
 - PBD 43,213
 - PWC 43,213
 - PWD 43,213
 - RAL 40,213
 - RAR 40,213
 - RBL 40,213
 - RBR 40,213
 - RET 11,36,79,141,211,213
 - RIA 37,215
 - RIB 37,215
 - RLA 39,215
 - RLB 39,215
 - RZA 37,215
 - RZB 37,215
 - SAL 40,215
 - SAM 38,215
 - SAP 38,215
 - SAR 40,208,215
 - SBL 40,216
 - SBM 38,216
 - SBP 38,216
 - SBR 40,92,208,216
 - SCD 85
 - SDC 46,216
 - SDI 47,148,149,216
 - SDO 47,148,149,216
 - SDS 46,85,216
 - SEC 39,216
 - SES 39,217
 - SFC 47,137,217
 - SFS 47,137,217
 - SIA 37,217
 - SIB 37,217
 - SLA 38,217
 - SLB 39,217
 - SOC 39,218
 - SOS 39,218
 - SSC 47,137,220
 - SSS 47,137,218
 - STA 34,218

STB 34,218
 SZA 37,218
 SZB 37,218
 TCA 35,219
 TCB 35,219
 WBC 43,219
 WBD 43,219
 WWC 43,219
 WWD 43,220
 XFR 34,146,220
 arithmetic 84
 entry 51
 groups 32
 operands 32
 patching 187
 processor 25
 pseudo-:
 General 18,51,54,223
 ANY 128
 BSS 56,57,75
 COM 59,128,129,195,196
 DAT 57
 END 5,11,17,55,59,195,271
 ENT 77,78,194
 EQU 59,75,197
 EXE 196
 EXT 59,77,78,80,195,197
 HED 64,65
 IFA 66
 IFB 66
 IFC 66
 IFD 66
 IFE 66
 IFF 66
 IFG 66
 IFH 66
 IFP 66
 LIT 74,196
 LST 61,62,65
 NAM 5,11,17,55,59,195
 REP 59
 SET 71,72,75,197
 SKP 62,63,64,65
 SPC 65
 SUB 11,59,78,108,128,129
 194,196
 UNL 61,62,65
 XIF 66,67
 non-listable 65
 repeating 59
 timing 221
 Int_to_rel utility 81,104

Integer Math group 35
 Integers:
 General 58,102,105,114,117,118
 octal 58
 structure 30
 Interfaces:
 General 133,134,147
 98032 (GPIO) 135,138,147,149
 237,239,241,244,247,250
 98033 (BCD) 239,244
 98035 (Clock) 237,239,257
 98036 (Serial) 237,239,241,244
 Interrupt I/O 133,138
 Interrupt service routines:
 General 21,138,140,149
 called from BASIC 150,151
 defined 5
 errors in 194
 linkage 141,152
 reserved symb 231
 reserved symbols 69
 state in 145
 Interrupts:
 General 209,221
 execution time 221
 lockout time 221
 related machine instructions 47
 Isr_access utility 81,143,149

j

JMP 36,211,221,222
 JSM 36,79,211,214,221,222

k

Keyboard 133
 Keys:
 CONTINUE 111,171,175
 RUN 111
 STEP 170,171,172,173,180,193
 STORE 11,14,53

I

LDA 34,211,213,221,222
 LDB 34,211,221,222
 LGT function 101,266
 LINES option, IASSEMBLE statement
 60,63,64
 LINK 53,268
 LIST option, IASSEMBLE statement
 60,61
 LIT 74,196,224
 LOAD 177
 LOG function 101,226
 LPY 221
 LST 61,62,65,224
 Labels:
 BASIC 50,52
 assembly 194,195
 assembly language 51,52
 Lines:
 Flag 47,217
 Status 218
 blank, in listings 65
 flag 137
 status 47,137,138
 Listing:
 General 61,62
 directives 60
 Literals:
 General 72,75
 as data generators 58
 evaluation of 72
 form of 72
 nesting 73
 nonsensical use of 74
 pools 74,196,271
 Load/Store group 34
 Lockout times:
 DMA 221
 interrupt 221
 Logical:
 group 41
 operations 41

m

MASS STORAGE IS 5,156
 MLY 45,211,221,222
 MOD operation 101
 MPY 35,212,222

MRX 44,87,89,91,212,221,222
 MRY 45,87,89,91,92,212,221,222
 MWA 46,90,213,221,222
 Machine address 78
 Machine architecture 25,26
 Machine code 33
 Maintenance agreements 273
 Mantissa shifting 44,45
 Manual:
 Assembly Language Quick Reference . 2
 Interfacing Concepts 134,135,136
 Mass Storage Techniques 5,17
 228,229
 Operating and Programming 17,21
 228,229
 structure 2
 Mass storage:
 General 8,22,53,56,156
 Descriptor (MSD) 157,160,161,163
 ROM 269
 Transfer Identifier (MSTID) .. 160–162
 Transfer identifier (MSTID) ... 157-159
 errors 269
 reading from 157
 unit specifier (msus) 5,157,163,269
 unit specifier (msus), defined 5
 writing to 160
 Memory:
 General 56
 dumps 181
 general organization 28
 map 27,232
 protected 28,178,180,193
 Mm_read_start utility 81,156–159
 Mm_read_xfer utility 81,156–159
 Mm_write_start utility 81,156,160,161
 Mm_write_test utility 81,156,160,161
 Modules:
 General 22
 creation 55
 defined 9
 names 17
 object 8,8,8
 object,defined 5
 source 5,5
 storage 56
 Multiplication:
 BCD 46,90,210
 binary 35,212

n

NAM 5,11,17,55,59,195,224
 NEXT 265
 NOP 47,213,221,222
 NOT operation 101
 NRM 45,89,213,221,222
 NUM function 226
 Names:
 modules 17
 Normalization 44,89
 Numbers:
 full-precision ... 58,78,84,102,103,104
 105,117,118
 full-precision,structure 30
 octal 184,225
 short-precision ... 58,78,103,105,114
 117,118
 short-precision,structure 30

O

OCT declaration 182
 OCTAL 16,101,169,184,193,229,270
 OFF INT 16,156,191
 ON ERROR 191,192
 ON INT 16,150
 ON declarations 267
 OPRM file-type 19,22,23
 OPTION BASE 265
 OR 101
 Object:
 code 7,8,9,14,60,68
 modules 8
 modules, defined 5
 Octal,expression,defined 5
 One's complement 41,208
 Operands 32
 Operating system 141
 Operations:
 AND 101
 EXOR 101
 Logical 41
 MOD 101,266
 NOT 101
 OR 101
 Output cycle, explained 25
 Overflow flag 35,38,39,145,180,186
 207,217,218,219
 Overlap mode 130

p

PAUSE key 170,171,172
 PBC 43,213,221,222
 PBD 43,213,221,222
 PI 101
 PRINT 183,186
 PRINT USING 268
 PROUND 101
 PWC 43,213,221,222
 PWD 43,213,221,222
 Page:
 format, listings 62
 headings, listings 64
 length, listings 64
 base 29,33,221
 base, defined 30
 current 29,33,221
 current, defined 30
 defined 29
 Parameters:
 blind 112
 in SUB pseudo-instruction 109
 Pausing 7,8
 Plotter ROM 268
 Pointers,stack 26,27,69,70,231
 Pools, literal 74,196,271
 Print_string utility 81,167
 Printer_select utility 81,166
 Priorities, for select codes 140
 Processors:
 Binary Processor Chip (BPC) ... 25,26
 Extended Math Chip (EMC) 25
 General 142
 Input-Output Controller (IOC) ... 25,26
 bus 25
 instructions 25
 Programmed I/O 138
 Programs:
 assembly language,developing 7
 counter 26,70
 counter,map 27
 creation 8,9,49
 defined 9
 entry 49
 stepping 7,8
 stepping through 170
 Protected memory 28,178,180,193
 Put_bytes utility 81,122,124,125,128
 Put_elem_bytes utility 81,122,125,128
 Put_element utility 81,122,123,125,128
 Put_file_info utility 81,163,165
 Put_value utility 81,122,123,124,128

r

RAL 40,40,213,221,222
 RAR 40,213,221,222
 RBL 40,213,221,222
 RBR 40,213,221,222
 REDIM 116
 REL 110,128,224
 REN command 266
 REP 59,224
 RES function 101
 RET 11,36,79,141,211,213,221,222
 RETURN 150
 RIA 37,215,222
 RIB 37,215,221,222
 RLA 39,215,221,222
 RLB 39,215,221,222
 RND function 101
 ROA 221
 ROMs:
 Assembly Development 1,2,193,261
 Assembly Execution 1,2,261
 Assembly Language 268
 I/O 268
 Mass Storage 269
 Plotter 268
 installation 3
 requirements of other 20
 RPT\$ 266
 RUN:
 command 20
 key 20,111
 RZA 37,215,221,222
 RZB 37,215,221,222
 Registers:
 General 26,33,37,38,40–44,48,70,135
 180,207,208–221,231
 BCD 84
 DMA 148,231
 DMA,General 26
 DMA,map 27
 I/O 26,27,70,187,231
 Peripheral Address 135,231
 arithmetic 70,92,93,94,95,231
 control 135
 external 27
 internal 26
 internal,map 27
 map 27
 preservation by ISRs 145
 stack 42
 status 136
 timing 221

Rel_math utility 81,99
 Rel_to_int utility 81,102
 Rel_to_sho utility 81,103
 Relocatable 68,75,196
 Rotation 40,215
 Routines:
 BASIC 175
 defined 9
 names 17

S

SAL 40,215,221,222
 SAM 38,215,221,222
 SAP 38,215
 SAR 40,208,215,221,222
 SAVE 267
 SBL 40,216,221,222
 SBM 38,216,221,222
 SBP 38,216,221,222
 SBR 40,92,208,216,221,222
 SCD 85
 SCRATCH A 21,143,159,162,194
 SCRATCH C 21,193
 SDC 46,216,221,222
 SDI 47,148,149,216,221,222
 SDO 47,148,149,216,222
 SDP 221
 SDS 46,85,216,221,222
 SEC 39,216,221,222
 SES 39,217,221,222
 SET 71,72,75,197
 SFC 47,137,217,221,222
 SFS 47,137,217,221,222
 SGN function 101
 SHO 110,128,196,224
 SIA 37,217,222
 SIB 37,217,221,222
 SIN function 101
 SKP 62,63,64,65,224
 SLA 38,217,221,222
 SLB 39,217,221,222
 SOC 39,218,221,222
 SOS 39,218,221,222
 SPC 65,224
 SQR function 101,266
 SSC 47,137,220,221,222
 SSS 47,137,218,221,222
 STA 34,218,221,222
 STB 34,218,221,222

- STEP key 170,171,172,173,180,193
- STOP 265
- STORE BIN 267
- STORE key 11,14,53
- STR 110,111,128,196,224
- SUB pseudo-instruction . 11,59,78,108,128
129,194,196,222
- SUBEND 150
- SUBEXIT 150
- SZA 37,218,221,222
- SZB 37,218,221,222
- Sales and Service offices 274
- Select codes, priorities 140
- Shift / Rotate group 40
- Shifting 207,211,212,215,216
- Shifting, mantissa 44,45
- Sho_to_rel utility 105
- Short-precision numbers 30,58,78,104
106,114,117,118
- Sign-magnitude format 85
- Signalling interrupts 151
- Skipping ... 37-39,46,47,209,216,217,218
- Source:
 - Source code:
 - Source,code,General 8,53,60,189
 - Source,listing control 61
 - Source,module 5
 - Space dependent mode 54
 - Stack group 42
 - Stack group, in ISRs 146
 - Stacks:
 - General 42,213,213,213,213,219
 - pointers:
 - General 26,69,70,231
 - map 27
 - registers 42
- Statements, BASIC:
 - ASSIGN 163,164,165
 - CALL 19,111
 - COM 115,128,195,267,270
 - DATA 266
 - DISABLE 156
 - EDIT 266
 - ENABLE 156
 - END 195,265
 - FOR 265
 - GET 53,177,268
 - IASSEMBLE ... 12,16,20,60,62,65,190
194,226
 - IASSEMBLE ALL 60
 - IBREAK 16,169,174,179,181,186
193,227,270
 - IBREAK ALL 169,178,227
 - IBREAK DATA 169,177,179,180
193,227
 - ICALL 12,16,18,19,107
108,115,117,123,131,172,227
 - ICHANGE 16,169,187,227
 - ICOM 12,16,18,19,21-23,193
196,227,270,271
 - IDELETE 16,18,22,227
 - IDELETE ALL 227
 - IDUMP .. 16,169,181,186,194,228,270
 - ILOAD 14,16,18,20,22,55,171
194,228
 - IMAGE 268
 - IMEM 228
 - INORMAL 16,169,179,193,228
 - IPAUSE OFF 16,169,174,228
 - IPAUSE ON 16,169,171,175,229
 - ISOURCE 11,49,50,53,54,63,229
 - ISTORE 14,16,19,23,55,194,229
 - ISTORE ALL 24
 - LINK 53,268
 - LOAD 177
 - MASS STORAGE IS 5,156
 - NEXT 265
 - OFF INT 16,156
 - ON ERROR 191,192
 - ON INT 16,150
 - ON declarations 267
 - OPTION BASE 265
 - PRINT 183,186
 - PRINT USING 268
 - REDIM 116
 - RETURN 150
 - SAVE 267
 - STOP 265
 - STORE BIN 267
 - SUBEND 150
 - SUBEXIT 150
 - Status line 47,218
 - Status registers 136
 - Stepping programs 7,8
 - Strings:
 - General 117,118
 - as data generators 117
 - structure 30
 - Subprograms, BASIC ... 155,163,176,179
 - Subprograms,errors 266
 - Subroutines 36,211,213
 - Substrings:
 - changing value of 125
 - retrieving 119,121
 - retrieving from arrays 121

Symbolic operations 69
 Symbols:
 General 196,197
 address of 185
 defining 71
 error regarding 194
 external 76
 pre-defined 69,153,231
 Synchronous access 142
 Syntax, fundamental 6

t

TAN function 101,266
 TCA 35,219,221,222
 TCB 35,219,221,222
 TYP function 101
 Tape cartridge 133,141
 Tape cartridge, Demonstration 261
 Ten's complement 45,46,86,208
 Test/Alter/Branch group 38
 Test/Branch group 37
 Timings:
 clock 221
 execution 221
 instruction 221
 lockout 221
 Transfers,DMA 149
 Two's complement 35,219
 Typing aids, demonstration cartridge 261

u

UNL 61,62,65,224
 Utilities:
 General 78,79,180,233
 Arithmetic 99
 Arithmetic,operand registers 27
 Busy 81,130
 Error_exit 81,186,191
 Execution of 172
 Get_bytes 81,116,119,128
 Get_elem_bytes 81,116,120,128
 Get_element 81,116,118,121,128
 Get_file_info 81,164
 Get_info 81,114,118,121,123,128
 Get_value 81,116,117,118,119,128
 Int_to_rel 81,104
 Isr_access 81,143,149

Mm_read_start 81,156,157,158,159
 Mm_read_xfer 81,156,157,158,159
 Mm_write_start 81,156,160,161
 Mm_write_test 81,156,160,161
 Print_string 81,167
 Printer_select 81,166
 Put_bytes 81,122,124,125,128
 Put_elem_bytes 81,122,125,128
 Put_element 81,122,123,125,128
 Put_file_info 81,163,165
 Put_value 81,122,123,124,128
 Rel_math 81,99
 Rel_to_int 81,102
 Rel_to_sho 81,103
 Reserved symbols 70
 Sho_to_rel 81,105
 Writing 231,235

v

Value checking 183
 Variables:
 General 56
 BASIC 13,175
 retrieving values from 117
 value checking 183

w

WBC 43,219,221,222
 WBD 43,219,221,222
 WWC 43,219,221,222
 WWD 43,220,221,222
 Word:
 General 43
 defined 5
 transfers 34,220

x

XFR 34,146,220,221,222
 XIF 66,67,224
 XREF option, IASSEMBLE statement 60

Your Comments, Please...

Your comments assist us in improving the usefulness of our publications; they are an important part of the inputs used in preparing updates to the publications.

Please complete the questionnaire, fold it up and return it to us. Feel free to mark more than one box to a question and to make any additional comments. If you prefer not to give us your name just leave the last part, name and address, blank. All comments and suggestions become the property of HP.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

HEWLETT-PACKARD COMPANY
DESKTOP COMPUTER DIVISION
3404 EAST HARMONY ROAD
FORT COLLINS, COLORADO 80525

ATTN: CONTROLLER DOCUMENTATION

FIRST CLASS
PERMIT NO. 37
LOVELAND, COLO.



Assembly Language ROM Errors

184	Improper argument in OCTAL or DECIMAL function
185	Break Table overflow
186	Undefined BASIC label or subprogram name used in IBREAK statement
187	Attempt to write into protected memory; or, attempt to execute instruction not in ICOM region
188	Label used in an assembled location not found
189	Doubly-defined entry point or routine
190	Missing ICOM statement
191	Module not found
192	Errors in assembly
193	Attempt to move or delete module containing an active interrupt service routine
194	Address out of range in IDUMP statement
195	Routine not found
196	Unsatisfied externals
197	Missing COM statement
198	BASIC's common area does not correspond to assembly module requirements
199	Insufficient number of BASIC COM items

Assembly-Time Errors

DD	Doubly-defined label
EN	END instruction missing; or module name does not match
EX	Expression evaluation error
LT	Literal pools full or out of range
MO	ICOM region overflow
RN	Operand out of range
SQ	Argument declaration pseudo-instruction out of sequence
TP	Incorrect type of operand used
UN	Undefined symbol

